2

NOTE DI SOFTWARE

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FPDM-108
RNSW-100

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e dell'Istituto di Cibernetica dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie o bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
La preparazione del testo nel formato adatto alla fotoriproduzione è a carico degli autori, ai quali verranno comunicate le norme redazionali.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione non vengono revisionati da un comitato tecnico. Pertanto, ogni contributo pubblicato riflette le opinioni personali dei suoi autori, e non necessariamente quelle del Comitato di Redazione.
Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
viene distribuito a chi ne faccia richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti
Redazione: Franco Soresini

NOTE DI SOFTWARE

2

FPDM-108

Aprile 1977

Indice:

QUESTO NUMERO	pag.	1
CONTRIBUTI TECNICI:		
- MACRO PROCESSORS: GENERAL CONCEPTS, di V. Cajani, R. Rousseau	"	2
- MACRO JCL, CONCETTI E APPLICAZIONI, di C. Stoppa	"	7
- USO DI UN MACROPROCESSOR PER LINGUAGGI AD ALTO LIVELLO. COBRA (COboI RAising): UN'APPLICAZIONE IN COBOL, di G. Bonesso, M. Maiocchi, D. Spadoni	"	10
- MACRO E ASSERZIONI, di G. Gobbi, L. Trabattoni	"	14
- MACRO PER VALUTAZIONI DI COPERTURA DI TESTS, di F. Gariboldi	"	18
- USO DI UN MACROGENERATORE PER LA GENERAZIONE DI PROGRAMMI DIAGNOSTICI SOTTO SISTEMA OPERATIVO, di M.L. Monopoli, F. Prigione	"	20
- L'USO DI MACRO JCL PER IL TESTING PERSONALIZZATO, di L. Trabattoni	"	27
- STRUTTURE DI DATI ASTRATTE IN ASSEMBLER MEDIANTE MACRO, di C.Barbaglio	"	29
- UN ESEMPIO DI MACRO PER GESTIRE STRUTTURE DI DATI IN ASSEMBLER: IL CASO DEGLI ALBERI, di F. Faidutti	"	33
- USO DI UN MACROGENERATORE NELLO SVILUPPO DI COMPILATORI FIRMWARE SPECIALIZZATI, di F. Prigione	"	37
- L'USO DI MACRO PER LA GESTIONE SEMIAUTOMATICA DI CONDIZIONI ECCEZIONALI: L'ESEMPIO DI DACBOL, di G. Zafferri	"	41
- STAND ALONE SYSTEM GENERATION LANGUAGE: LA GENERAZIONE DI UNA "IMMAGINE DI MEMORIA" ATTRAVERSO LA DEFINIZIONE DI MACRO, di C. Monducci, M. Parini	"	44
- LA COSTRUZIONE DI MACROPROTOTIPI STRUTTURATI, di M. Maiocchi, R. Polillo	"	47
AVVENIMENTI	"	52
BIBLIOGRAFIA: I macrogeneratori	"	54
IL SEGNALIBRO	"	55

Autorizzazione del Tribunale di Milano n. 87 del 23 febbraio 1977

QUESTO NUMERO . . .

Il secondo numero di NOTE DI SOFTWARE ha carattere monografico. Esso raccoglie un insieme di lavori che, sia pure sotto differenti angolazioni, si riferiscono tutti all'uso di un macro-generatore nell'attività di produzione di software. Le esperienze illustrate, anche se centrate su un prodotto specifico -il macroprocessor del Livello 62- coprono un campo di applicazioni alquanto vasto: dalla estensione di un linguaggio di programmazione all'attività di prova di un prodotto software, alla costruzione di compilatori, ecc..

I primi due lavori hanno carattere introduttivo. Il primo (Cajani e Rousseau) illustra i concetti fondamentali connessi all'uso di un macro-generatore di tipo 'general purpose' e, più in particolare, del macro-generatore del Livello 62. Il secondo (Stoppa) costituisce invece una introduzione all'uso di macro di Job Control Language. Anche in questo caso, gli esempi si riferiscono al Livello 62.

I lavori che seguono hanno carattere più specifico. Barbaglio e Faidutti, in due lavori coordinati, trattano dell'uso di macro per la gestione di strutture di dati in un linguaggio assembler (NAL). Diverse esperienze sull'uso di macro nell'attività di test vengono illustrate da Gobbi e Trabattoni (macro per "asserzioni", che espandono codice per il controllo, al momento dell'esecuzione, di condizioni specificate dal programmatore), da Gariboldi (strutture di controllo di tipo "flaggante"), da Monopoli e Prigione (macro per comporre programmi di tipo diagnostico), e infine da Trabattoni (uso di macro di JCL per la personalizzazione di catene di test). L'articolo di Prigione mostra quindi come un compilatore, con l'uso di macro, possa essere reso parametrico rispetto al linguaggio sorgente e al linguaggio oggetto, mentre Monducci e Parini descrivono la generazione, mediante macro, di un'immagine di memoria di supporto a un emulatore. L'uso di un macroprocessor in un linguaggio ad alto livello (il Cobol) viene esemplificato da Bonesso, Maiocchi e Spadoni (il sistema COBRA) e da Zafferri (il trattamento di condizioni di errori che sorgono dopo l'esecuzione di verbi di I/O). Maiocchi e Polillo, infine, forniscono un insieme di indicazioni per la programmazione strutturata di macroprototipi.

A questi lavori si aggiunge una breve bibliografia, diretta a chi desideri un inquadramento di tipo introduttivo ai macrogeneratori.

La Redazione

CONTRIBUTI TECNICI

MACRO PROCESSORS: GENERAL CONCEPTS

V. Cajani (°), R. Rousseau (+)

1. INTRODUCTION

The main drive behind computer development has been due to man's desire to make the machines do some of the more tedious aspects of his work. Likewise, the software development specialist has been impelled to seek means to liberate himself of the task of writing repetitive elements, which are time consuming, though unfortunately necessary, in the production of software.

The concept of "macroization" (or writing macros) was born from the simple desire to find short cuts or abbreviations in coding. This article shall limit itself to the discussion of macros for software production, although it is clear that macro techniques are not limited to this area, but are applicable wherever large volumes of répetitive elements are a normal part of production.

The primary use of macros currently is to make implementation languages more flexible and to tailor them to the implementor's needs. The original goals have become secondary. This is due to the evolution of high level languages which now contain elements (e.g., the COPY verb in standard ANSI COBOL) which aid the programmer in finding "short cuts" and "abbreviations". The new goals are aimed more towards specific applications than towards specific machines or operating systems. Flexibility, while a valid goal for languages of all levels, is particularly interesting for assembly level languages; reflecting the growing need of software specialists to communicate with one another using abbreviations and acronyms representing new level of abstractions in the evolution of software.

In our opinion this is the most revolutionary and significant impetus towards the evolution of Macro Processors (the programs which process macros), and will be discussed in detail below while other areas are merely noted. They include:

- text editing
- language translation .

Macro Processors generally operate in conjunction with a base language. The level 62 Macro Processor for example is predisposed to compile macros for the NAL assembler and the COBOL compiler. Input to the Macro Processor is a source program which includes primitive statements of the base language and Macro Processor statements. Output from the Macro Processor is a source program which contains only elements of the base language. The Macro Processor recognizes the elements in the input which it must treat. These elements are commonly known as "macro calls". The treatment which is known as "generation" involves the substitution of the call with the code which has been furnished the Macro Processor at some previous time. Essentially a Macro Processor performs a replacement function. The saved code is known as a macro "body".

(°) H.I.S.I. - Pregnana Milanese
(+) INTER-ACT - Milano

2. MACRO PROCESSOR LOGICAL TIMES

It is evident in the following illustration that there are different times in the preparation of a program. They are:

Time 1 - The user decides what extensions to the source language are necessary in order to write his program in a more efficient way. He then choses the set of macro statements to be implemented. This interval is part of the overall analysis time of the program.

Time 2 - The macro bodies are defined and saved in a reserved area for the Macro Processor.

Time 3 - Programs are written containing the language extensions in the form of macro calls.

Time 4 - The calls are expanded by the Macro Processor and base language elements are produced. This is also known as Generation Time.

Time 5 - The program is compiled or assembled and object code is produced.

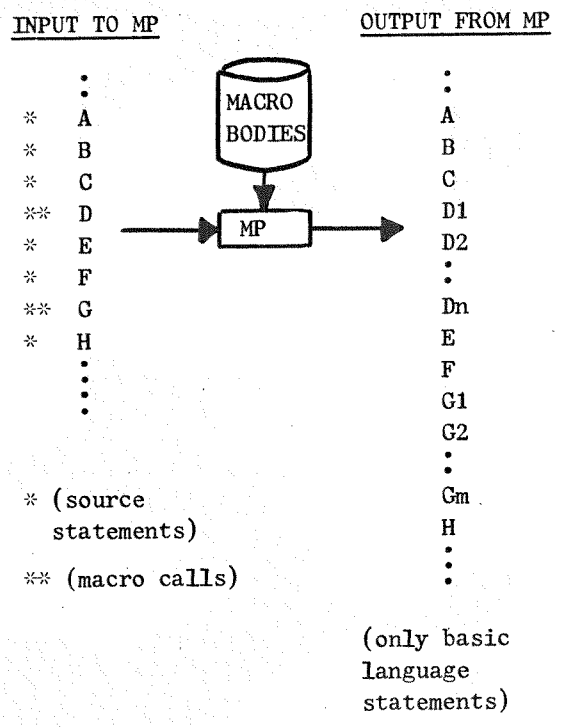


fig. 1

3. ELEMENTARY LANGUAGE ASPECTS

An extension to the base language is formalized in terms of Macro Processor statements written in a block of code called a macro body.

The most elementary of all statements is that which always causes the generation of a piece of text (or a string of characters).

Additional statements are provided to give more flexibility in text generation. These statements represent a logical flow when seen at generation time. In fact macro bodies may be flowcharted analogously to programs, showing the possible outputs according to the different conditions under which the expansion is requested.

Events conditioning the text generation may be:

- local to the body processing
- inherent to the macro call to this macro body
- global to the source input (interbody communication).

The above events are handled by the basic instructions of the Macro Processor.

To clarify the concepts of Macro Processing, examples and references will be made using the Honeywell 62 Macro Processor. However, the general concepts are valid for all Macro Processors.

The 62 Macro Processor unlike some others contains a set of instructions which must be visible at body definition time. It is not possible to generate them dynamically. This goal may be achieved, however, with successive passes through the Macro Processor.

Nested Calls. A nested call is one embedded in a macro body which may be processed when the Macro Processor expands the host macro. Clearly the nested call has meaning only when control is passed to that point during expansion. If the nested call name happens to coincide with the host macro name, it becomes a recursive macro call. Since a macro call is a means by which a sentence (and not only a verb) is written in the base language, nesting extends this facility without limitation.

Control Statements. Events may be tested during the expansion of a macro body using control statements. Then, based on the results, control of the Macro Processor is passed to certain points within the body.

Variables. Variable fields may be defined to save and process global or local information. Local variables are elements defined and used only within a specific body. Global variables once defined (i.e., once executed) are known throughout the life of the Macro Processor run. This latter facility provides the means to pass information from one expansion to another.

Parameters. Parameters are those elements within a macro body which require actualization at expansion time. The actual values are furnished in the macro call. In some predefined cases, if no value is furnished, the Macro Processor inserts a default value which is furnished at definition time.

4. CALLING A MACRO PROTOTYPE

A macro call is a new syntactical statement of the base language. The form is free, but contains 2 basic elements. The first element identifies the macro body requested; the elements which follow (called actual arguments) provide the attributes to the new statement. The general form is:

$statement-name attribute-1,attribute-2...;

In the level 62 Macro Processing language, all macro calls start and end with the characters "$" and ";", respectively.

The attributes are of two basic types:

- positional arguments
- keyword arguments

A macro call may contain any combination of the above 2 types. The number and type are defined in the macro body. A positional argument is one which must appear in a specified position of the macro call. A keyword argument (which appears after all the positional arguments are given) may appear in any order, but is known by a specified identifier and a value. The value of a keyword argument may be the identifier itself (in this case it is called a self-identifying keyword).

5. ELEMENTS OF A MACRO BODY

The statements which constitute a macro body are enclosed between the two statements:

- DEF
- END

The DEF statement has 3 basic functions:
- to start the macro body
- to name the macro body
- to define the parameters (into which are mapped the arguments) used in the macro body

The END statement is used to terminate the macro body.

The elements contained within the DEF-END enclosure define a macro program and provide the basis for the processing by the Macro Processor when a call is found.

Other statements in the Macro Processing language are:

1- generate string
2- GLOBAL, used to define a global item
3- LOCAL, used to define a local item
4- SET, used to set a global or local item
5- IF, used to test events in the macro body
6- GOTO, used to transfer control during expansion time
7- VALUE, used to put variable values into the output text
8- %n, used to put parameter values into the output text.

Statement 1 above is actually implicit for the 62 Macro Processor. Any element within a body which is not statements 2 through 8 is automatically compiled for generation mode. Statements 7 and 8 also create generation mode instructions. However, the Macro Processor must first substitute a specific element prior to generation. Statements 2 through 6 do not cause any generation but create instructions to perform a variety of processing in the macro body.

It should be kept in mind that these instructions are compiled at macro body definition time and are only executed at expansion time.

Following is an example of a macro prototype body which when called expands into an Identification Environment Division in a COBOL program.

```
(1) $DEF IDEV(NAME AUTH=NIL,DATE=NIL,
        REM=NIL,SEC=NIL);-
(2) '         IDENTIFICATION DIVISION.'
    '         PROGRAM-ID'.
(3)             %NAME .
(4) $IF %AUTH EQ '' THEN $GOTO LAB1;;-
            AUTHOR.
              %AUTH .
(5) $LAB1:-
            INSTALLATION.
              CHAMPOLUC.
    $IF %SEC EQ '' THEN $GOTO LAB2;;-
            SECURITY.
              %SEC .
    $LAB2:-
    $IF %DATE EQ '' THEN $GOTO LAB3;;-
    '         DATE-WRITTEN.'
              %DATE .
    $LAB3:-
    $IF %REM EQ '' THEN $GOTO LAB4;;-
            REMARKS.
              %REM .
    $LAB4:-
            ENVIRONMENT DIVISION.
            CONFIGURATION SECTION.
    '         SOURCE-COMPUTER.'
    '           LEVEL-62.'
    '         OBJECT-COMPUTER.'
    '           LEVEL-62.'
(6) $END;-
```

Note:

- Line (1) defines the macro name and the parameters of the body, NAME is a positional parameter which has no default value and therefore requires a value at call time. AUTH,DATE,REM,SEC are keyword parameters which have default values of NIL
- Line (2) is a simple generation element
- Line (3) is an example of an element which requires the actualization of a parameter
- Line (4) is a conditional statement which tests if a parameter value is furnished in the call. If not, a jump is made to LAB1. If furnished control passes to the next statement
- Line (5) is a label for passage of control (see line 4 above)
- Line (6) is the end of the definition

6. HOW THE MACRO PROCESSOR MAY BE USED

1. A frequent use of the level 62 Macro Processor is to generate pieces of code common to different software elements. For example,
   a) Global data of multiple procedures or programs. The use of the Macro Processor assures that whenever a change is made, all the macroized pieces are changed automatically. A new logical element is thus introduced (e.g., $GLOB; in which case a global block is defined without the necessity of defining the individual elements of the area).
   b) Interfacing points between user software and the operating system and/or Data Management permitting interfacing on a logical level irrespective of the specific code generated. A high level of independence is achieved. The day that the interface changes, the source programs need only be recompiled (it is clear that only the macro body is modified).

2. Another use is to generate families of homogeneous programs. For example,
   a) Application packages may be tailor made to user needs. This reduces the need for general purpose packages capable of keeping everyone happy, but generally being memory guzzlers and time consumers. The Macro Processor can produce a custom version producing more efficient object code. Fewer decisions at run time imply fewer instructions executed and thus faster execution time.
   b) Test program generation is an essential aid to the debugging of large systems and compilers. Software implementors need the facility to produce large numbers of similar programs, varying elements of the programs so as to cover a complete zone of testing for their products. The Macro Processor can easily be used to satisfy this need.

3. The most powerful tool which is offered is that which extends a language tailored to the user's needs. In this area the programmer need not code at an elementary level but at a logical functional level (where the function is meaningful in terms of the application).

   An actual case experienced by the writer involved the implementation of IDS for the GE 635. It was necessary to add new verbs to the standard COBOL repertoire. The absence of a Macro Processor required the creation of a preprocessor which involved 3 man years (1966).

7. CONCLUSION

The principal obstacle to a more widespread use of Macro Processors has been the low performances at generation time. Most Macro Processors must not only search for and evaluate the macro call on a character by character basis, but must dynamically interpret the macro body in the same way. The strategy used in the level 62 Macro Processor has been to minimize this process by precompiling the macro body. By requiring that all statements be visible at macro definition time, it is possibile to strip down the body breaking it up into elements called pseudo instructions. These pseudo instructions are actually commands to the Macro Processor. The Macro Processor also makes use of a free work stack area (the size of which is declared by the user) keeping the maximum amount information in memory at a time (eliminating the need to constantly reload from an auxiliary memory macro bodies). These factors along with the fixing of the column in which a macro call starts have contributed to the production of a Macro Processor which when executed is more often I/O bound instead of CPU bound.

Due to its high performance and flexibility the level 62 Macro Processor has allowed the introduction of advanced programming tools at a minimal implementation cost. Structured Programming is one of the most significant examples. The implementors were concerning only with the logical and external linguistic aspects of the application without the need to implement a precompiler or compiler to process it.

8. REFERENCES.

(1) Brown,P.J., "Macro Processors" (1974) John Wiley & Sons

(2) Brown,P.J., "Extending high-level languages by macros-a practical case evaluation" Proceedings of Software 72 Conference, Transcripta Books, London 95-99 (1972)

(3) Ferguson,D.E., "The evolution of the meta-assembly program", Comm. ACM, 9, 3, 190-193 (1966)

(4) Kent,A., "Assembly-language macroprogramming: a tutorial oriented toward the IBM 360", Computing Surveys, 1,4, 183-196 (1969)

(5) Rousseau,R., "Honeywell Level 62 Macro Processor functional specifications", (documento interno H.I.S.I. MN157) (1973)

(6) Strachey,C., "A general purpose macrogenerator", Computer Journal, 8,3, 225-241 (1965)

(7) Stout, R. (ACT), "HELP/REBEL", ACT Publishers, N.Y. (1975)

MACRO JCL, CONCETTI ED APPLICAZIONI

C. Stoppa - H.I.S.I. - Pregnana Milanese

1. INTRODUZIONE

Il Job Management, negli elaboratori elettronici, tende a diventare tanto più sofisticato quanto maggiore é la versatilità delle macchine progettate. La loro architettura si modifica molto rapidamente nel tempo in funzione di nuove necessità di Data Management, di minori tempi di risposta e di più elevata automazione di procedura di calcolo. A ciò deve essere aggiunto l'utilizzo di una gamma sempre maggiore di periferiche diverse verso cui procedure di calcolo devono essere trasparenti e l'uso contemporaneo di una stessa risorsa in fase di multiprogrammazione.

Un contributo alla soluzione di una problematica così vasta è stato possibile grazie all'introduzione di linguaggi ad alto livello, i Job Control Languages, in grado di permettere un'efficiente comunicazione uomo/macchina al fine di stabilire un Job Flow in funzione del verificarsi di determinati eventi, di associare risorse ai singoli step esecutivi, di dichiarare il grado di possibile coutilizzo di una o più risorse per due o più step in esecuzione parallela.

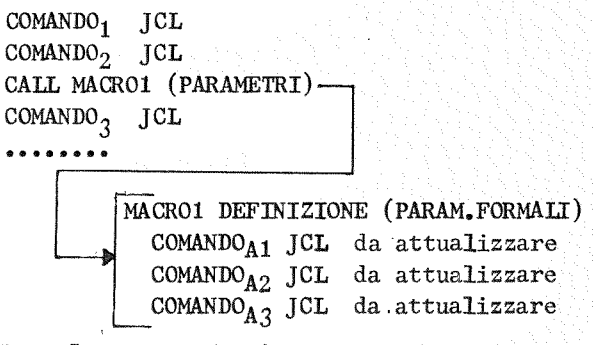
Un Job Control Language comporta però problemi comuni a tutti i linguaggi ad alto livello, vale a dire: problemi di leggibilità, affidabilità, portabilità, ridondanze. Per tali motivi e tenuto conto che nell'elaborazione dell'utente molti step di lavoro vengono eseguiti un numero rilevante di volte in condizioni analoghe a meno di pochi elementi di variazione, le tendenze attuali dei costruttori sono quelle di offrire all'utente mezzi più potenti di comunicazione che, utilizzati nell'ambito dell'Input Stream (descrizione dei lavori da eseguire tramite comandi JCL), permettano di risolvere agevolmente i problemi citati. Tali mezzi sono generalmente noti con il nome di Macro JCL.

Ciascuna Macro è composta da una sequenza logica di statement, residenti in un file di disco, i contenuti dei quali possono essere attualizzati in funzione delle esigenze del momento dell'uso. Il richiamo di una macro è effettuato tramite un comando di "Macro Call" che può prevedere un numero variabile di parametri destinati all'attualizzazione dei comandi costituenti il corpo della macro.

I sistemi più sofisticati prevedono chiamate in cascata di macro fino ad un livello di sprofondamento massimo legato alla filosofia d'impostazione dello specifico JCL. Le macro possono essere in parte fornite direttamente dal costruttore, per la gestione di particolari programmi di sistema di utilità, ed in parte costruite dall'utente stesso a fronte delle proprie esigenze.

L'esplosione di una macro può essere realizzata tramite un Macroprocessor o tramite un Interpretatore/Processatore; il risultato finale è comunque l'integrazione del corpo della macro attualizzato nell'input stream secondo lo schema seguente:

Prima della Macroespansione:

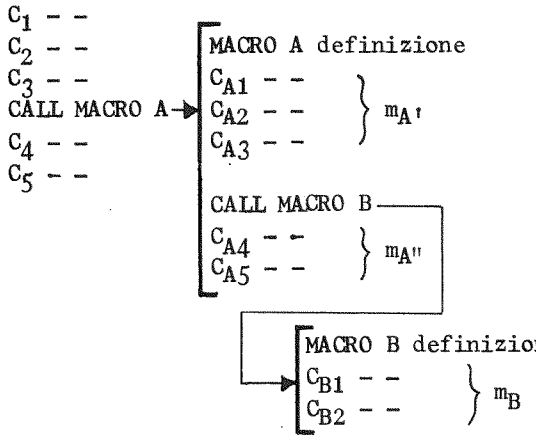


Dopo la Macroespansione:

COMANDO$_1$ JCL
COMANDO$_2$ JCL
COMANDO$_{A1}$ JCL attualizzato
COMANDO$_{A2}$ JCL attualizzato
COMANDO$_{A3}$ JCL attualizzato
COMANDO$_3$ JCL

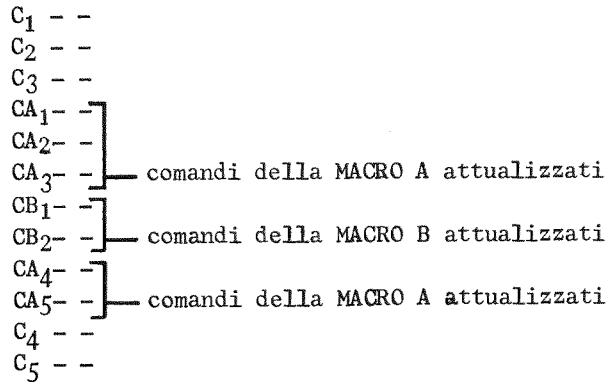
Il caso delle macro chiamate in cascata viene risolto analogamente.

Prima della Macroespansione:



dove $m_{A'}$ e $m_{A''}$ sono statement della MACRO A ed m_B della MACRO B da attualizzare.

Dopo la Macroespansione:



Il Macroprocessor o l'Interpretatore/processatore JCL effettuano le attualizzazioni dei comandi secondo tecniche comuni a tutti i tipi di Macroprocessor. Più precisamente l'attualizzazione è realizzata a fronte di un carattere speciale seguito da un nome definito nello statement di macrodefinizione cui fa riscontro un parametro attuale dichiarato nello statement di chiamata della macro.
Un breve esempio chiarirà meglio il concetto.

```
$COMANDO  .... ;
$COMANDO  ....;
$CALL MACRO A (PAR1=3,PAR2=ALFA);
$COMANDO  ....;
```

```
$DEF MACRO A (PAR1=X,PAR2=Y)
$COMANDO   CAMPO1=&PAR1;
$COMANDO   CAMPO2=&PAR2;
```

Dopo l'espansione della MACRO A l'input - stream avrà il seguente formato:

```
$COMANDO  ....;
$COMANDO  ....;
$COMANDO  CAMPO1=3;
$COMANDO  CAMPO2=ALFA;
$COMANDO  ....;
```

2. ESEMPLIFICAZIONE APPLICATA AL LINGUAGGIO JCL HISI L62.

Gli esempi che seguono mostrano la validità dell'applicazione delle macro nel sofisticato linguaggio JCL degli elaboratori Honeywell serie L62.

Si supponga di dover eseguire il programma CARDISK che legga schede da Lettore (di sistema), elabori i dati e registri i risultati su un file di nome EXFILEA su un disco chiamato VOL1. Gli statement di JCL che l'utente deve introdurre nell'elaboratore per comunicargli la richiesta di elaborazione sono:

```
......
......
$STEP CARDISK;
$ASSIGN CARD, @ IN;
$ASSIGN OUTFILE,EXFILEA,MEDIA=VOL1;
$ENDSTEP;
......
```

Qualora lo stesso programmino debba essere eseguito più volte pretendendo l'input da file sempre diversi ed emettendo l'output pure su file diversi, l'utente sarebbe costretto ad introdurre nell'Input Stream una serie di descrizioni di step (quattro statement/step) per adeguarsi alle esigenze.
Ciò può tradursi in un onere tanto più pesante quanto più complicata è la descrizione di ogni singolo step. La leggibilità dell'Input Stream verrebbe compromessa. La probabilità di errore verrebbe esaltata.
Una notevole semplificazione sarebbe ottenuta se l'utente avesse preventivamente scritto ed inserito nell'apposita libreria la seguente Macro il cui nome sia MACRO1:

```
$MAC ID=(PAR=P,FILEI=XXX,MEDIAI=VVV,
        FILEO=YYY,MEDIAO=ZZZ);
$STEP CARDISK;
$ASSIGN OUTFILE,&FILEO,MEDIA=&MEDIAO;
$WHEN '&PAR',=,'CARD',JUMP=LABEL1;
$ASSIGN INFILE,&FILEI,MEDIA=&MEDIAI;
$JUMP LABEL2;
LABEL1 $ASSIGN CARD,@ IN;
LABEL $ENDSTEP;
```

Infatti, all'utente basterebbe introdurre i seguenti comandi nell'Input Stream primari a quando voglia eseguire diverse volte il programma CARDISK con diverse caratteristiche di input/output:

```
....
$MACRO1 PAR=CARD,FILEO=EXFILEA,MEDIAO=VOL1;
$MACRO1 PAR=NOCARD,FILEI=INFILEA,MEDIAI=VOL2,
        FILEO=EXFILEB,MEDIAO=VOL3;
$MACRO1 .....;
$MACRO1 .....;
```

per far sì che l'elaboratore processi automaticamente la seguente Input Stream:

```
....
....
$STEP CARDISK;
$ASSIGN OUTFILE,EXFILEA,MEDIA=VOL1;
$ASSIGN CARD,@IN;
$ENDSTEP;
$STEP CARDISK;
$ASSIGN OUTFILE,EXFILEB,MEDIA=VOL3;
$ASSIGN INFILE,INFILEA,MEDIA=VOL2;
$ENDSTEP;
$STEP CARDISK;
....
....
$ENDSTEP;
$STEP CARDISK;
....
....
```

Un secondo esempio può evidenziare il beneficio ottenibile tramite una macro in scelte alternative di esecuzione. Si supponga sia stata inserita in libreria la seguente macro dal noma MACRO2:

```
$MAC  ID=(EXEC=C,FILE1=FILE1,-
MEDIA1=MEDIA1,FILE2=FILE2, MEDIA2=MEDIA2);
$WHEN '&EXEC',=,'B',JUMP=STEP1;
$STEP CARDISK;
$ASSIGN CARD;@IN;
$ASSIGN OUTFILE,&FILE1,MEDIA=&MEDIA1;
$ENDSTEP;
$WHEN '&EXEC',=,'A',JUMP=FINE.
STEP1 $CONTINUE;
$STEP DISKPR;
$ASSIGN INFILE,&FILE2,MEDIA=&MEDIA2;
$ASSIGN PRINTER,@OUT;
$ENDSTEP;
FINE $CONTINUE;
```

L'utente avrà la possibilità di fare eseguire il programma CARDISK o DISKPR o ambedue secondo i parametri espressi nello statement di macro call inserito nell'Input Stream primaria. Infatti si possono avere i seguenti casi:

. Esecuzione del solo CARDISK

```
....
$MACRO2 EXEC=A,FILE1=EXFILEA,MEDIA1=VOL1;
....
```

verrà automaticamente processata la seguente Input Stream

```
....
$STEP CARDISK;
$ASSIGN CARD, @IN;
$ASSIGN OUTFILE,EXFILEA,MEDIA=VOL1;
$ENDSTEP;
....
```

. Esecuzione del solo DISKPR

```
....
$MACRO2 EXEC=B,FILE2=INFILEB,MEDIA2=VOL2;
....
```

verrà automaticamente processata la seguente Input Stream

```
....
$STEP DISKPR;
$ASSIGN INFILE,INFILEB,MEDIA=VOL2;
$ASSIGN PRINTER, @OUT;
$ENDSTEP;
....
```

. Esecuzione di ambedue i programmi

```
....
$MACRO2 EXEC=C,FILE1=EXFILEA,MEDIA1=VOL1,
        FILE2=INFILEB,MEDIA2=VOL2;
....
```

verrà automaticamente processata la seguente Input Stream

```
....
$STEP CARDISK;
$ASSIGN CARD, @ IN;
$ASSIGN OUTFILE,EXFILEA,MEDIA=VOL1;
$ENDSTEP;
$STEP DISKPR;
$ASSIGN INFILE,INFILEB,MEDIA=VOL2;
$ASSIGN PRINTER, @ OUT;
$ENDSTEP;
....
```

L'esemplificazione fatta mette chiaramente in evidenza l'utilità delle MACRO JCL l'uso delle quali diventa tanto più conveniente quanto più complicata è la descrizione dei Job. Infatti, risulta immediatamente percepibile l'aspetto positivo rispetto alla leggibilità, l'affidabilità, la portabilità, la ridondanza. Naturalmente l'utente non deve fare abuso di macro, il che può tradursi in una degradazione delle performance per via del tempo impiegato nell'esplosione, ma trovare un giusto equilibrio che gli permetta di massimizzare il throughput globale.

USO DI UN MACROPROCESSOR PER LINGUAGGI AD ALTO LIVELLO. COBRA (COBol RAising): UN'APPLICAZIONE IN COBOL

G. Bonesso (°)- M. Maiocchi (+) D. Spadoni (°)

1. INTRODUZIONE

La possibilità di innalzare il livello del linguaggio di programmazione per utenti COBOL é fornita sull'H62 dall'accoppiamento del macroprocessor al compilatore COBOL, nel prodotto denominato COBRA.

Il progetto COBRA si inquadra in un processo evolutivo delle tecniche implementative del software che ha visto un progressivo innalzamento del livello dei linguaggi di programmazione necessario per seguire il rapido diffondersi del computer da pochi centri specializzati a gruppi sempre più vasti di end-users.

Lo sviluppo del software ha infatti condotto l'utente a potersi servire dell'elaboratore senza essere costretto a conoscerne la struttura interna. Si é passati così dai linguaggi machine-oriented a quelli che, come il COBOL, racchiudono in una singola frase di evidente significato intere sequenze elementari.

Con l'introduzione di COBRA si é compiuto un ulteriore passo in questa direzione: in un macroprototipo si possono infatti considerare racchiuse più frasi di un linguaggio (sia istruzioni che definizioni di dati) ed il suo richiamo può essere assunto come un nuovo verbo di livello più elevato.

L'impiego di un macroprocessor dà rispetto a quello di un precompilatore alcuni vantaggi. Sul piano implementativo non richiede modifiche del compilatore, mentre sul piano applicativo offre la più completa portabilità: l'output del macroprocessor é infatti un programma completamente standard e perciò trasportabile con costi limitatissimi su qualsiasi compilatore.

(°) H.I.S.I. - Pregnana Milanese
(+) Istituto di Cibernetica
Università di Milano

2. FUNZIONALITA' DI COBRA

Le funzionalità offerte all'utilizzatore di COBRA si possono articolare in tre aspetti essenziali:

a) Macroprototipi realizzati dall'utente.
questo punto é già stato esaminato in precedenza.

b) Macro fornite all'utente.
saranno messe a disposizione dell'utente alcune macro residenti su libreria di sistema che permettono di risolvere facilmente alcuni problemi di codifica.
Il numero e le specifiche funzionali di tali macro sono tuttora oggetto di studio e l'attenzione si é fino ad ora focalizzata sui seguenti argomenti:
   1) MOVE con indicazione della lunghezza e/o a partire da displacement specificato.
   2) Swapping su condizione di due campi assegnati.
   3) Scansione right - left.
   4) Generazione parametrica di Identification ed Environment Division.
   5) Generazione standard di strutture di dati.

c) Macro per la programmazione strutturata.
queste macro permettono di applicare la programmazione strutturata in COBOL.

I criteri applicati nella scelta delle strutture di controllo, ispirandosi all'esperienza di STNAL, hanno condotto alla realizzazione di tre tipi di strutture molto generalizzate che assolvono completamente ed anzi integrano le funzionalità offerte dalle varie strutture di STNAL.
Si é scelto di ridurre il numero dei tipi di strutture arricchendo la duttilità di ciascuna per rendere più semplice e più leggibile il programma e per facilitare il compito del programmatore con una maggiore sinteticità di concetti: si ha così a disposizione, infatti, un set di strutture per la selezione, uno per l'iterazione e

uno per la selezione diretta di eventi.
Si é inoltre particolarmente curata la diagnostica, controllando e segnalando anche gli errori di nidificazione di strutture diverse tra di loro (in tal modo vengono rilevate situazioni errate come

```
$CIF ...
   $DO ...
$ENDIF ...
   $REPEAT ...)
```

e verificando, mediante una macro al termine del programma, che tutte le strutture siano debitamente chiuse.

La diagnostica é integrata: i messaggi di errore sono trattati direttamente dal compilatore e vengono emessi insieme e con lo stesso formato di quelli Cobol.

Un'estensione alle funzionalità offerte dal set di macro per la programmazione strutturata é rappresentata dall'espansione opzionale di ulteriore codice che permette, come FLAGGER per programmi NaL [1] , di riconoscere e memorizzare i cammini percorsi durante il lancio del programma. Questo permette di valutare l'efficienza di catene di tests di un programma, assicurandone la completa copertura. [2]

Riportiamo qui la sintassi delle macro messe a disposizione

1. Selezione

```
$CIF 'condizione Cobol',THEN;
...
$ELSE[IF 'condizione Cobol'];
...
$ENDIF;
```

La selezione ad "alternative parallele" permette di integrare la funzione della struttura CASE nella precedente.

2. Iterazione

```
$DO   [{WHILE 'condizione Cobol'                    }]
      [{VARYING=id1,FROM=id2,TO=id3                 }]
                                  ,BY=id4     ;
$EXIT   [IF='condizione Cobol'] ;
$REPEAT  [UNTIL='condizione Cobol'];
```

3. Selezione diretta di eventi

```
$EXECUTE UNLESS=(lista di eventi)
                                [,LOOP]    ;
$EVENT nome [,IF='condizione Cobol'] ;
$THEN;
$WHEN nome [,EVENT=nome]  ;
$ENDEXEC;
```

3. ESEMPIO DI PROGRAMMA COBOL GENERATO TRAMITE RICHIAMI DI MACRO

Notazione contratta:

```
$IDEV SKITRIP,AUTH=AAP,DATE=010176,REM=UPDATES-INVENTORY-FILE;
$SEL PRINTFILE,PRINTER;
$SEL DISKFILE,DISK,DVC=MSU310,ORG=REL,AMODE=RANDOM,RLKEY=DISK-KEY,LAST;
$FD DISKFILE,BSIZE=16,DR=(REC1,REC2,REC3);
$RD3 FIELD1A;
$RD2 FIELD1,P=X(7);
$RD3 FIELD2,P=X(10);
$RD2 FIELD3,USE=3,P=S9(5)V99;
$RD2 FIELD4,P=XX,LAST;
      .
      .
      .
WORKING-STORAGE SECTION.
01  A.

$GLOBAL AAA=PIC S9(7)V9(2) OCCURS 2 USAGE IS COMP-3;
     02  A1 $$AAA;
     02  A2 $$AAA;
     02  A3 $$AAA;
      :
 PROCEDURE DIVISION.

$PERFORM LABEL=(LAB1=A,LAB3=B,LAB7=C),ON=CTR;
      :
      :
```

Notazione estesa:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                                    $IDEV
    SKITRIP.
AUTHOR.
    ASSOCIATION OF ANIMAL PROTECTORS
INSTALLATION.
    CHAMPOLUC.
DATE-WRITTEN.
    1 JANUARY 1976.
SECURITY
    LOOSE.
REMARKS.
    UPDATES-INVENTORY-FILE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
    LEVEL-62.
```

```
INPUT-OUTPUT SECTION.                          $SEL
FILE-CONTROL.
    SELECT PRINTFILE ASSIGN TO PRINTER.
```

```
    SELECT DISKFILE ASSIGN TO DISK-MSU310
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM
    RELATIVE KEY IS DISK-KEY.                  $SEL
DATA DIVISION.
FILE SECTION.
```

```
FD   DISKFILE
     BLOCK CONTAINS 16 RECORDS                 $FD
     LABEL RECORDS ARE STANDARD
     DATA RECORDS ARE  REC1  REC2  REC3.
```

```
01   REC1.
     02 FIELD1.                          $RD2/$RD3
     03 FIELD1A PIC IS X(7).
     03 FIELD2  PIC IS X(10).
     02 FIELD3  PIC IS S9(5)V99 COMP-3.
     02 FIELD4  PIC IS XX.
          .
          .
          .

WORKING-STORAGE SECTION.
01  A.
  02  A1 PIC S9(7)V9(2) OCCURS 2
                         USAGE IS COMP-3.
  02  A2 PIC S9(7)V9(2) OCCURS 2
                         USAGE IS COMP-3.
  02  A3 PIC S9(7)V9(2) OCCURS 2
                         USAGE IS COMP-3.
       .
       .
       .
PROCEDURE DIVISION.
```

```
      IF CTR = 1 PERFORM A GO TO FINE.
      IF CTR = 3 PERFORM B GO TO FINE.
      IF CTR = 7 PERFORM C GO TO FINE.
FINE.                                          $PERFORM
```

4. ESEMPIO DI GENERALIZZAZIONE DI MOVE A LUNGHEZZA VARIABILE CON DISPLACEMENTS SPECIFICATI.

Notazione contratta:

```
 WORKING-STORAGE SECTION.
$WORKMOVE;

 PROCEDURE DIVISION.

$MOVE FIELD1,FIELD2,DISP1=4,DISP2=3,LL=5;
```

Notazione estesa:

```
 WORKING-STORAGE SECTION.
 77  WORK1 PIC X(256).
 77  CTR1  PIC 9(3).
 77  CTR2  PIC 9(3).
 77  CTR3  PIC 9(3).
 77  DEL   PIC X  VALUE ""9A"".

 PROCEDURE DIVISION.

     MOVE 4 TO CTR1
     MOVE 3 TO CRT2
     UNSTRING FIELD1 INTO WORK1 WITH
                          POINTER CTR1.
     MOVE 6 TO CTR3
     STRING DEL DELIMITED BY SIZE INTO
                          WORK1.
     WITH POINTER CTR3.
     STRING WORK1 DELIMITED BY DEL
                          INTO FIELD2
     WITH POINTER CTR2.
```

Un ulteriore sviluppo di questa funzionalità potrebbe essere rappresentato dall'introduzione di un controllo di overflow e quindi di un nuovo parametro della MOVE del tipo OVF= imperative statement.
Inoltre il carattere esadecimale usato come delimiter può essere scelto dall'utente mediante un parametro opzionale della $WORKMOVE del tipo DEL = character.

5. ESEMPIO DI USO DI MACRO PER LA PROGRAMMAZIONE STRUTTURATA.

```
*   STORHAND: SUMMARIZE STORE MOVEMENTS.
  STORHAND.
       OPEN OUTPUT PRINTER.
       OPEN INPUT MOVMTS
       MOVE TITLE TO PRINT-TITLE.
       WRITE PRINT-TITLE.
       PERFORM READMOVM.
 $DO  WHILE='EOF IS NOT EQUAL TO "E"';
       PERFORM ELABART THRU END-ELABART.
 $REPEAT;
       MOVE NUMPARTS TO PR-TOT
       MOVE PARTS-AFFECTED TO PR-AFFECTED.
       WRITE PRINTOT AFTER ADVANCING 3 LINES.
       CLOSE PRINTER MOVMTS.

       STOP RUN.
* ------------------------------------------------
  READMOVM.
       READ MOVMTS AT END
       MOVE "E" TO EOF
       MOVE 0 TO MOVKEY.
* ------------------------------------------------
*   ELABART: SUMMARIZES THE MOVEMENTS OF THE
*   SAME ARTICLE  AND  PRINTS THE RELATIVE
*   RESULTS.

  ELABART.
       MOVE MOVKEY TO STORKEY.
       ADD 1 TO NUMPARTS.
       MOVE 0 TO NETMOVM.
 $DO;
       PERFORM ELABMOVM THRU END-ELABMOVM.
 $REPEAT UNTIL='MOVKEY NOT EQUAL TO STORKEY';
       MOVE ARTICLE TO PR-STOP.
       MOVE NETMOVM TO PR-NET.
       WRITE PRINT-ART.
  END-ELABART.
       EXIT.
*------------------------------------------------
*   ELABMOVM: SUMMARIZES THE QUANTITY IN
*   NETMOVM AND READS A NEX RECORD.

  ELABMOVM.
 $CIF 'MOVCODE = "I"',THEN;
       SUBTRACT MOVQTITY FROM NETMOVM.
 $ELSE;
       ADD MOVQTITY TO NETMOVM.
 $ENDIF;
       PERFORM READMOVM.
  END-ELABMOVM.
       EXIT.
```

6. RIFERIMENTI

[1] F. Gariboldi, B. Giordani, G.A. Lanzarone, P.V. Renesto: "FIAGGER, Uno strumento per la valutazione dei risultati delle prove." Note di Software N. 1

[2] F. Gariboldi: "Macro per valutazione di copertura di tests." Presente numero di Note di Software

[3] G. Bonesso, D. Spadoni: "Structured programming in Cobol F.S." documento interno HISI, Pregnana nov. 1976.

[4] Proceeding of Giornate di Studio su: "Programmazione Strutturata: Esperienze ed Orientamenti", Milano 23 / 25 giugno 1976.

MACRO E ASSERZIONI

G. Gobbi(°), L. Trabattoni(°)

1. INTRODUZIONE

Le asserzioni permettono sia di migliorare la qualità e la leggibilità dei programmi, sia di ridurne i costi di collaudo e di manutenzione.

Se le asserzioni sono scritte in forma processabile, le condizioni in esse specificate sullo stato del programma possono essere verificate durante l'esecuzione, con conseguente segnalazione delle violazioni e delle informazioni di rilievo.

In una vasta gamma di applicazioni l'uso di macro offre uno strumento economico e flessibile per la scrittura e l'elaborazione delle asserzioni. Vengono qui presentati alcuni esempi.

2. UN'ASSERZIONE E'

un'affermazione, relativa a un punto del programma, sullo stato corrente e sugli stati passati del programma, che riteniamo vera tutte le volte che il controllo giunge a quel punto.

Esempi:

2.1 - 3 schede lette da INFILE e poste in CARD1, CARD2, CARD3.
2.2 - Esiste uno ed un solo valore di XO per cui STAT(XO)='READY'
2.3 - (TABLE(I) = KEY) OR (HI-LO > 0 AND I=0)
2.4 - Z = MAX(X,Y)
2.5 - COM-TABLE :[Table of valid commands], FROM INIT-TABLES
2.6 - ASSERT LOCAL (P; A*; V)* (4)
2.7 - ASSERT LOCAL S=Y.3 & Y=X.3; (4)
2.8 - ASSERT ORDER (A(*,3)) ASCENDING (5)

3. LE ASSERZIONI SERVONO

- a facilitare il disegno del programma:

  documentano i dati di input e di output di ogni modulo, specificandone il significato, i requisiti, lo stato, le relazioni con altri moduli ("DA" o "A"); rendono possibile il controllo di effettiva coerenza delle interfacce (1)

  documentano le assunzioni che il programmatore sta facendo e le rendono esplicite (2)

  documentano le condizioni critiche e gli invarianti che devono essere mantenuti in caso di modifiche al programma (2)

- a documentare il programma:

  ai commenti sulle motivazioni del programma, e al codice relativo, le asserzioni aggiungono informazioni utili a capire lo stato della computazione nei punti significativi del programma (3)

  sono destinate a chi deve rivedere il disegno e l'implementazione (auditors), a chi deve manutenere il programma, a chi ne deve studiare o imitare il disegno e la realizzazione (3,5)

- come strumento di debugging:

  come commento inserito nel codice: sono un ausilio essenziale sia nelle fasi di controllo preliminare (ispezioni al disegno e al codice, walkthrough) sia nella ricerca degli errori durante le fasi di testing

  come testo processabile: la scrittura delle asserzioni in forma processabile e la relativa elaborazione permettono di segnalare durante l'esecuzione del programma tutte e sole le violazioni alle condizioni

---

(°) H.I.S.I. - Pregnana Milanese

previste, e di ottenere, in corrispondenza, informazioni sullo stato del programma.

4. IL DEBUGGING

ed il testing, benchè le tecniche di programmazione strutturata possano ridurre radicalmente il numero di errori, non sono eliminabili.

Uno dei problemi principali è individuare gli errori sulla base di informazioni insufficienti o sovrabbondanti (2).

- Talvolta nei punti critici del programma vengono inserite parti di codice per il controllo dello stato del programma (valori di variabili, flags, indici etc.). Tuttavia per controllare e segnalare violazioni di condizioni previste è necessario aggiungere al programma strutture di dati e istruzioni destinati solo a tale scopo. Queste aggiunte, difficili da separare dal resto del programma, forniscono informazioni spesso insufficienti, nascondono la logica del programma e sono difficili da deattivare dopo la fine del testing (4).
- Viceversa l'uso del tracing fornisce una massa voluminosa ed indiscriminata di dati, di faticosa interpretazione e di nessun beneficio per la documentazione del programma. La limitazione della quantità di dati prodotti dal tracer richiede una selezione accurata delle variabili da controllare e dei punti di accensione e di spegnimento del tracer. Il tracing è relativamente costoso da implementare (modifica l'interpretazione delle istruzioni di assegnamento) e causa una degradazione di prestazioni nel programma in esecuzione, prolungando i tempi di turn-around (compilazione, esecuzione, diagnostica, correzione) e di testing (2,4).

5. L'ELABORAZIONE DELLE ASSERZIONI

fornisce un potente contributo a rendere più facili ed economici il debugging, il testing e la manutenzione dei programmi.

Consideriamo ad esempio la seguente asserzione, scritta in forma di macro-call all'interno di un programma Cobol:

```
$ASSERT "TABLE(I) = KEY",V1="TABLE(I)",
        V2=KEY;
```

possiamo immaginare di segnalare a run time le violazioni dell'asserzione emettendo su console, terminale o file di stampa un messaggio del tipo:

```
ASSERTION 5 :[TABLE(I) = KEY] FALSE
10TH EVALUATION - 2ND VIOLATION
VALUE OF VARIABLES:
  TABLE(I) = BROWN
  KEY = BROWNS
```

La disponibilità di uno strumento automatico per la segnalazione degli errori a run time incoraggia l'uso delle asserzioni, con conseguente miglioramento della qualità del disegno e della documentazione programmi.

Le asserzioni nella forma

ASSERT <espressione booleana>

possono essere elaborate sia con l'introduzione nel linguaggio di programmazione del verbo ASSERT (2), sia tramite preprocessor (precompilatore o macroprocessor); nel primo caso il compilatore dispone di tutte le informazioni relative ai dati che compaiono nell'asserzione; il secondo caso è più flessibile perchè non richiede modifiche al compilatore.

Le asserzioni possono essere inserite o eliminate senza alterare il comportamento del programma (ammesso che l'espressione booleana non invochi funzioni con effetti collaterali) (2).
Una facile selezione a compile time o a preprocessor time può escludere del tutto o in parte l'elaborazione delle asserzioni, presenti in tal caso come commenti.

Le asserzioni in forma processabile assolvono alla loro funzione di "guardiani" della correttezza del programma per tutta la vita del programma, segnalando nei casi di modifiche o correzioni l'insorgere di condizioni anomale o di effetti collaterali.

6. L'USO DI MACRO

è un modo flessibile ed economico per la scrittura e l'elaborazione delle asserzioni.
Seguono alcuni esempi che presentano la forma della chiamata a una macro ASSERT e un breve riassunto delle funzionalità.

Esempio 6.1: Controllo e segnalazione di violazione dell'asserzione.

```
$ASSERT 'PAGE-PH-SIZE <64';
```

quando l'asserzione non è verificata, viene emesso su console il messaggio seguente, che chiede all'operatore di scegliere se continuare o terminare l'esecuzione:

```
 QQQQ ASS. 2 :[PAGE-PH-SIZE < 64] FALSE
 QQQQ ENTER CONTINUE OR STOP
*QQQQ ?
??
*QQQQ CONTINUE
```

Esempio 6.2: Segnalazione della violazione e stampa dei valori di variabili del programma.

```
$ASSERT 'EOF-FLAG = 1',V1=LAST-PAGE-FLAG,
       V2=EOF-FLAG;
```

Quando l'asserzione non è verificata, viene emesso il messaggio seguente:

```
 QQQQ ASS. 15 :[EOF-FLAG = 1] FALSE
 QQQQ VALUE OF VARIABLES:
 QQQQ   LAST-PAGE-FLAG = 0
 QQQQ   EOF-FLAG = 0
 QQQQ ENTER CONTINUE OR STOP
*QQQQ ?
??
*QQQQ STOP
```

Esempio 6.3: Segnalazione della violazione; stampa dei valori di variabili; stampa del numero di valutazioni e di violazioni dell'asserzione. Si può inoltre chiedere o che il programma termini dopo un numero di violazioni MAXV o che dopo ogni violazione venga eseguita la procedura specificata dal parametro ONVIOL.

```
$ASSERT 'EOF-FLAG = 1',V1=LAST-PAGE-FLAG,
       V2=EOF-FLAG,MAXV=10;

$ASSERT 'EOF-FLAG = 1',V1=LAST-PAGE-FLAG,
       V2=EOF-FLAG,ONVIOL=MONITOR;
```

Quando l'asserzione non è verificata, viene emesso il messaggio:

```
 QQQQ ASS. 4 :[EOF-FLAG = 1] FALSE
 QQQQ 2ND EVALUATION - 1ST VIOLATION
 QQQQ VALUE OF VARIABLES:
 QQQQ   LAST-PAGE-FLAG = 0
 QQQQ   EOF-FLAG = 0
```

Esempio 6.4: Valutazione delle asserzioni solo in alcuni blocchi del programma.

Si possono usare macro del tipo:

```
$BEGIN block-name;

$END block-name;
```

per generare il codice di delimitazione dei blocchi e macro variabili che descrivano la struttura ad albero del programma.

```
$SELECT BLOCK1=name1[,...[,BLOCKN=nameN]]
       [,IF='macro-condition'];
```

se macro-condition (espressione logica valutata a macroprocessor time) è vera o assente, vengono valutate le asserzioni nei blocchi specificati.

$$\$SELECT\ LEV1=\left\{\begin{matrix}ALL\\int\end{matrix}\right\}[,\ldots[,LEVN=\left\{\begin{matrix}ALL\\int\end{matrix}\right\}]]$$

[,IF='macro-condition']; (int 1)

se macro-condition è vera o assente, vengono valutate le asserzioni appartenenti ai blocchi la cui posizione nell'alberatura del programma è specificata dai parametri LEV1,...,LEVN.

$SELECT LEV1=1,LEV2=3,LEV3=ALL; provoca la valutazione delle asserzioni nei blocchi rappresentati con tratto continuo in figura 1.

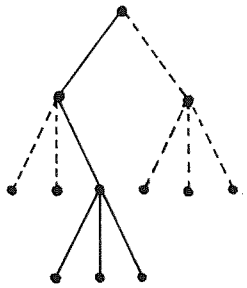


fig. 1

Esempio 6.5: Selezione a run time delle asserzioni da valutare.

```
$ASSERT 'TERMINAL-STATUS = OK',
       IF='MODE = INTERACTIVE';
```

A run time l'asserzione viene valutata se e solo se MODE = INTERACTIVE.

7. ESTENSIONI

alle funzionalità presentate sono (2,4,5):

- Segnalazione automatica dei valori delle variabili (2)
- Asserzioni globali e asserzioni locali (4,5)
- Asserzioni sui dati (5), es.2.8
- Asserzioni sulla sequenza di esecuzione di classi di istruzioni (4), es.2.6
- Asserzioni sugli stati passati del programma (4), es.2.7
- Codice utente di supporto alla valutazione delle asserzioni (4)

8. RIFERIMENTI BIBLIOGRAFICI

(1) G. Gobbi, M. Maiocchi
"Uno standard di codifica ed autodocumentazione"
Università di Milano - Honeywell I.S.I.
Gennaio 1977

(2) D. Matusek
"The case for the assert statement"
Sigplan Notices, Agosto 1976

(3) G. Degli Antoni, G. Gobbi
"Aspetti relativi alla documentazione dei programmi"
Giornate di studio su "Programmazione strutturata: esperienze e orientamenti"
Milano, 23-24-25 Giugno 1976

(4) T. S. Chow
"A generalized assertion language"
2nd International Conference on Software Engineering
San Francisco 13/15 Ottobre 1976

(5) L. G. Stucki, G. L. Foshee
"New assertion concepts for self-metric software validation"
International Conference on Reliable Software
Los Angeles, 21/23 Aprile 1975

MACRO PER VALUTAZIONI DI COPERTURA DI TESTS

F. Gariboldi
H.I.S.I. - Pregnana Milanese -

1. IL TESTING DEI PROGRAMMI

Il controllo della qualità di un programma consiste nella verifica della "copertura" delle funzionalità dello stesso. Questa si ottiene mediante l'esecuzione del programma con differenti insiemi di dati, in modo da esercitare tutte quelle funzionalità che l'utente si aspetta.

Il metodo normalmente usato é quello delle "checklist", lista delle funzionalità per ciascuna delle quali viene individuato o progettato un test. Tale approccio risente però del fatto che la "checklist" è frutto dell'intuito di chi compila la lista delle funzionalità da verificare, che non é generalmente possibile mettere in relazione con le caratteristiche fisiche del programma o il disegno dello stesso.Occorre quindi un ulteriore affinamento del controllo di qualità di un programma, che si può ottenere col metodo della "copertura topologica".

Esso consiste nella verifica che tutti i possibili rami di un programma siano stati percorsi almeno una volta, intendendo per "ramo" ogni porzione di programma costituita da sequenze di istruzioni che non contengano istruzioni di controllo condizionato.

Ulteriori specificazioni sulle motivazioni del metodo e su altre esperienze esistenti si trovano in (1).

Questo metodo é particolarmente utile e significativo per il testing di programmi strutturati: infatti in questi casi la struttura fisica del programma riflette generalmente la struttura logica della computazione che viene eseguita, facendo fortemente convergere la copertura topologica verso la copertura funzionale.

2. MACRO PER IL CONTROLLO DELLA COPERTURA TOPOLOGICA

Le strutture di controllo per la programmazione strutturata in ambito assembler NAL e COBOL sono state realizzate, nei laboratori HISI di Pregnana Milanese,per mezzo di un insieme di macroprototipi.

Le macro che realizzano le strutture di controllo sono gli unici elementi che debbano venire controllati nella loro esecuzione per ottenere la copertura topologica del programma.

Da questo punto di vista é quindi sufficiente che ogni macro, su richiesta, espanda non soltanto il codice per realizzare la struttura di controllo, ma anche del codice aggiuntivo per poter individuare e registrare sugli elementi di un vettore, in memoria, il numero di volte che i rami della struttura vengono eseguiti. Ulteriori specificazioni si trovano in (2).

I costi dell'implementazione del codice aggiuntivo e del suo uso sono molto ridotti:

- dal punto di vista implementativo si richiede la costruzione, per l'inserzione nella zona dati del programma, di una macro ($FLAG) che genera un vettore i cui elementi vengono usati come contatori per le strutture di controllo; tale macro inoltre notifica a tutte le macro delle strutture di controllo presenti nel programma la richiesta di espandere il codice di "flagging".
  Un'altra macro ($ENDFLAG), inserita immediatamente prima dell'istruzione di fine programma, richiama una procedura la quale, la prima volta che il programma viene eseguito, copia il vettore su un flusso di disco, mentre le volte successive aggiorna il flusso sommando i contenuti degli elementi del vettore a quelli precedentemente registrati.Si ha così la possibilità di conoscere il grado di copertura topologica realizzata da una catena di tests lanciando un programma il quale, analizzando il flusso ed il sorgente, stampa il listing dello

stesso con riportato a fianco il numero di volte che le strutture di controllo sono state eseguite.

- l'occupazione di memoria del programma instrumentato aumenta mediamente di circa 1/3. Questo incremento di memoria non costituisce un problema, essendo l'instrumentazione richiesta una sola volta nella vita di un prodotto, quando cioè si vuol misurare il grado di copertura che i suoi test garantiscono.
- l'aumento del tempo di esecuzione é generalmente trascurabile.

4. CONSIDERAZIONI

L'uso delle macro con "flagging" incorporato fornisce una misura della qualità dei test eseguiti, in quanto tanto maggiore é la percentuale dei rami sollecitati, tanto migliore é il testing effettuato. Inoltre il listing "flagged" del programma é molto utile per il disegno e la realizzazione di nuovi tests, essendo facilmente rilevabili i rami del programma che non sono mai stati sollecitati.

RIFERIMENTI

(1) G.F. Fait, G.A. Lanzarone, L. Ferri: "A simple tool for automatic measuring of software testing". Proocedings of Human Engineering Conference, Milano 13 - 15 nov. 1974, pp. c189-c200.

(2) A. Brioschi, F. Gariboldi, M. Maiocchi: "A consistent set of tools for testing and structured programming". Honeywell Software Productivity Simposium, Minneapolis 26 - 28 apr. 1977.

USO DI UN MACROGENERATORE PER LA GENERAZIONE DI PROGRAMMI DIAGNOSTICI SOTTO SISTEMA OPERATIVO

M.L. Monopoli(°), F. Prigione(°)

1. INTRODUZIONE

Sono argomento di questa nota i programmi diagnostici delle periferiche del calcolatore Honeywell Level 62.

Tali programmi sono eseguibili in ambiente di multiprogrammazione - in concorrenza con i programmi User - e sono controllati da un Supervisore Diagnostico (DIOS) integrato nel sistema operativo GCOS 62.

I programmi in argomento, scritti in linguaggio "assembler", sono stati realizzati facendo ricorso a specifiche strutture funzionali e a specifiche sovrastrutture di controllo ottenute mediante l'uso del Macroprocessor Level 62.

2. MOTIVAZIONI

I programmi diagnostici costituiscono una "collezione" di programmi omogenei, nel senso che hanno comuni obiettivi e comune utenza (il field engineering, o l'operatore del centro di calcolo).

In virtù di questo fatto, il loro sviluppo deve assecondare caratteristiche quali :

- uniformità di disegno;
- standardizzazione delle norme d'uso;
- semplificazione della scrittura;
- rispetto scrupoloso dell'"integrity" del sistema.

3. STRUTTURA FUNZIONALE DEI PROGRAMMI DIAGNOSTICI

Composizione dei programmi. Tutti i programmi diagnostici sono sostanzialmente realizzati facendo ricorso al tipo di struttura rappresentato in Fig. 1.

La struttura rappresentata è di tipo filiforme, e costituita da sottostrutture iterative con ricorrenza variabile in funzione del raggruppamento di appartenenza.

Le operazioni diagnostiche. Le funzioni diagnostiche indicate nello schema di Fig. 1 rappresentano la composizione "standard" di un test diagnostico, il quale, a sua volta, è l'unità di composizione del programma diagnostico.

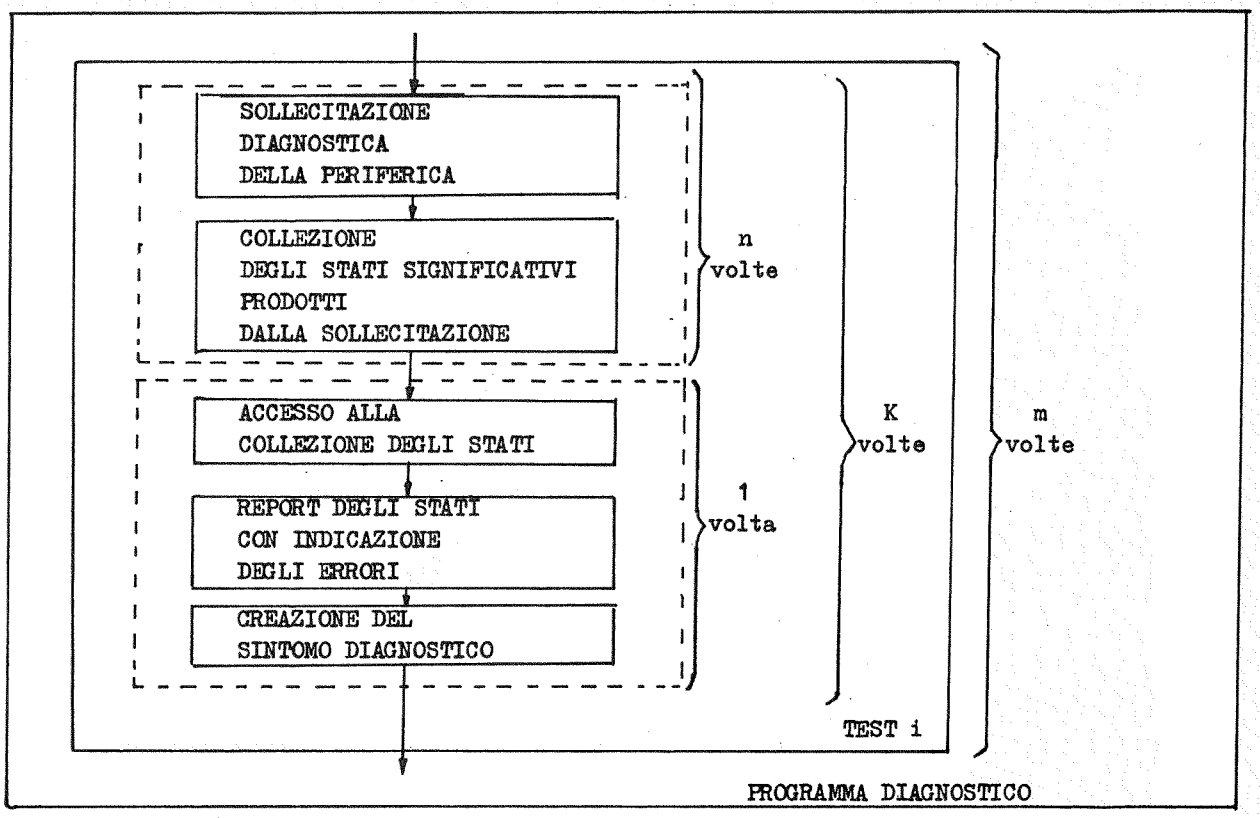
Il processo di base di un test consiste nell'inviare una serie di sollecitazioni alla periferica (accessi di input/output) che vengono eseguite ricorrendo, se necessario, anche all'uso di un componente di "firmware" specializzato alla diagnostica (FIDIB).

Le informazioni che ne sono il risultato vengono collezionate ed elaborate al fine di consentire l'individuazione della funzione logica o del componente fisico che danno errore.

Le indicazioni che tale elaborazione fornisce sono sostanzialmente di due tipi :

- un "report" di stampa, che associa ad ogni test e ad ogni punto di controllo l'indicazione dei risultati corretti o errati (per i Programmi Diagnostici di tipo "Funzionale")
- un "sintomo" generato dell'esecuzione di tutto l'insieme dei tests (il "Sintomo" è il risultato dell'elaborazione della collezione delle risposte alle sollecitazioni) (per i programmi Diagnostici di tipo ITR Isolation Test Routines).
  Il confronto fra il "sintomo" generato e una "biblioteca di sintomi" (parte integrante del programma diagnostico ITR) fornisce l'indicazione del package fisico che dà errore. La "biblioteca di sintomi" viene costruita mediante la "simulazione fisica" dei guasti più probabili (si creano cioè fisicamente i guasti, e, ad uno ad uno, se ne colleziona il sintomo, generando così la "biblioteca di sintomi").

(°) H.I.S.I. - Pregnana Milanese



STRUTTURA FUNZIONALE DI UN PROGRAMMA DIAGNOSTICO

FIG. 1

Realizzazione della struttura "funzionale" di controllo. La struttura "funzionale" di un programma diagnostico è costituita da un insieme di strutture di controllo, generate dal Macrogeneratore, che realizzano le specifiche funzioni diagnostiche di base (in modo equivalente a strutture STNAL quali if-then-else oppure while-do).

Descriviamo ora alcune delle "strutture di controllo" diagnostiche, e in particolare quelle che risolvono le principali "funzioni diagnostiche di base". Esse operano in modo controllato tanto nei confronti delle funzioni di sistema operativo quanto nei confronti delle collezioni di dati, e garantiscono pertanto che i requisiti di "integrity" siano soddisfatti.

A) $H_TDCHP (SOLLECITAZIONE DIAGNOSTICA)
Realizza l'esecuzione della operazione di Input/Output sulla periferica sotto diagnosi. Opera in funzione del contenuto di un campo di interfaccia (che definisce il tipo di operazione, il valore del "Time-Out" di protezione ....).

Ha formato :

$H_TDCHP	nome del flusso associato alla periferica in diagnosi , numero dei buffers utilizzati dall'operazione ;

B) $H_TDCKS e $H_TDCKB (COLLEZIONE DEGLI STATI SIGNIFICATIVI PRODOTTI DALLA SOLLECITAZIONE)
Gestiscono i risultati della sollecitazione, siano essi "stati di periferica" o "contenuti di buffers", e scrivono il "file" che costituisce la "collezione degli stati significativi".

Hanno il seguente formato :

$H_TDCKB	identificativo , nome dell'area contenente i risultati , termine di confronto ,

lunghezza del confronto;

termine di confronto = nome dell'area contenente i "valori attesi"

$H_TDCKS identificativo , nome del flusso associato , nome dell'area contenente i risultati, termine di confronto , lunghezza del confronto ;

termine di confronto = nome dell'area contenente i "valori attesi" e la maschera di selezione dei valori significativi.

C) $H_TDREP (REPORT DELLA COLLEZIONE DEGLI STATI)

Questa funzione, sulla base delle informazioni lasciate dalle strutture $H_TDCKS o $H_TDCHB , determina le condizioni di errore, genera il "sintomo diagnostico" relativo, stampa su console - se richiesto - i messaggi di controllo e/o di errore.

Ha formato :

$H_TDREP identificatore del report , tipo di sintomo da creare ;

tipo di sintomo da creare = quantità di informazione da porre nel"sintomo" (1 bit o n bits per ogni frase di controllo).

La struttura di composizione di un programma diagnostico, realizzato mediante l'uso di strutture di controllo, diviene pertanto essa stessa una "struttura controllata", essendo un insieme a diversi livelli di raggruppamenti di "strutture di controllo" di base.

La fig. 2 rappresenta la realizzazione mediante macro della struttura funzionale indicata in fig. 1.

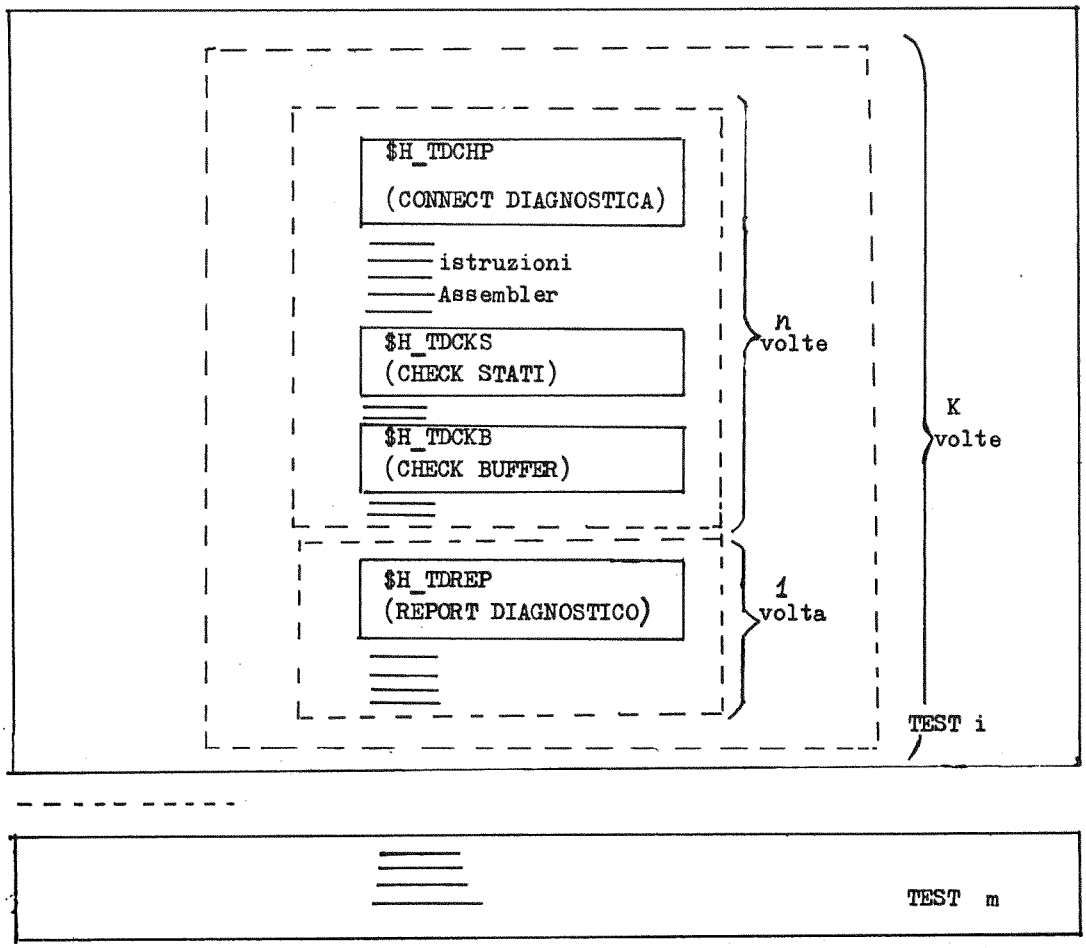


FIG. 2

4. STRUTTURA DI CONTROLLO DELLA ORGANIZZAZIONE E DELL'ESECUZIONE DEI PROGRAMMI DIAGNOSTICI

Struttura della Unità di Compilazione diagnostica. Tutti i programmi diagnostici, costituiti da un insieme di n "unità di programma" (test), hanno esigenze comuni per quanto riguarda sia l'occupazione di memoria che le modalità di esecuzione e sono pertanto realizzati mediante Unità di Compilazione aventi una struttura del tipo indicato in Fig. 3

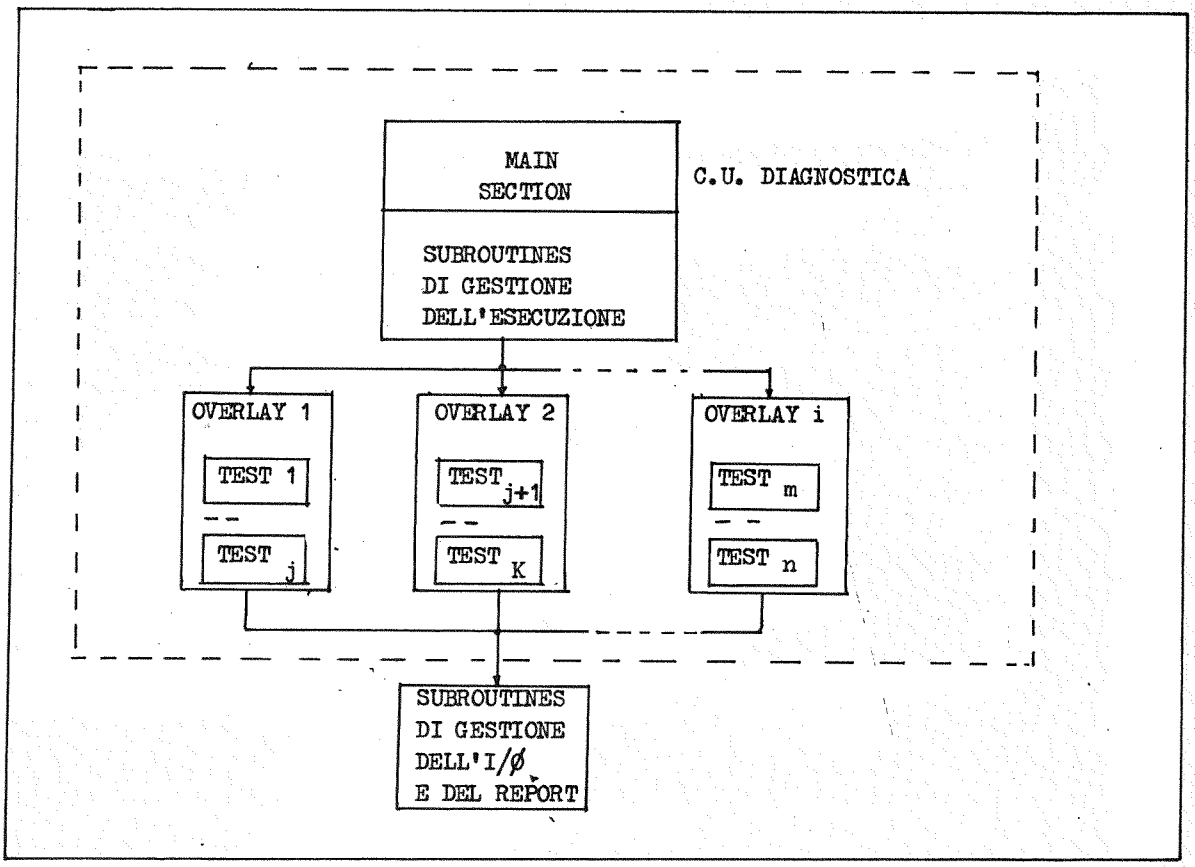


FIG. 3

Realizzazione della sovrastruttura di organizzazione della "composizione in memoria". La "composizione" in memoria dei programmi diagnostici è realizzata mediante frasi dichiarative rivolte al Macrogeneratore.

In questo modo il numero degli "overlays" e il numero di tests che li costituiscono possono essere dichiarati prima della sezione esecutiva del programma.

Per il fine in argomento, le principali strutture utilizzate sono le seguenti :

A) $H_TDJDEF Definisce il nome della "Unità di Compilazione" da generare ed il numero di "overlays" di cui si vuole sia composta.

Ha formato :

$H_TDJDEF	nome della procedura , numero di overlays ;

B) $H_TDNTST Definisce il numero di tests componenti ogni overlay.

Ha formato :

$H_TDNTST	numero di tests del primo overlay, numero di tests del secondo overlay , numero di tests dell'n-simo overlay ;

Realizzazione della sovrastruttura di controllo dell'esecuzione. Nella struttura dell'"Unità di Compilazione" viene creata, invocando una serie di "strutture di controllo" generate dal Macrogeneratore, una sezione principale che consente l' esecuzione controllata di una serie di tests variabile e parametrica, secondo dichiarazioni da associare a "run-time".

Tale sezione costituisce in linea di principio un "mini-sistema operativo interattivo diagnostico". Essa infatti, operando su un programma diagnostico "virtuale", inteso cioè composto da una serie di tests "connessi in modo virtuale", realizza "run-time" le connessioni che definiscono il programma che deve essere eseguito.

Nell'esempio di fig. 4, il programma diagnostico, composto virtualmente dalla serie TEST 1, TEST 2, TEST 3, TEST 4, TEST 5, TEST 6, TEST 7, TEST 8, viene invece eseguito, dopo il "linking di run time", come un programma così composto :

TEST 2 - TEST 3 - TEST 1 - TEST 5 - TEST 7 - TEST 4

Le principali "strutture di controllo" utilizzate per realizzare questa funzione sono di seguito riportate :

A) H_TDSEQ Genera la lista di identificazione dei tests che consentirà il "linking di run time" mediante il colloquio di console.

B) H_TDEXEC Realizza il "mini-sistema operativo interattivo" per la gestione dei tests a "run - time".

C) H_TDTST Fissa il punto di inizio del test.
Genera, quando necessario, in base alla descrizione di composizione degli overlays, il punto di inizio della sezione secondaria contenente n tests.

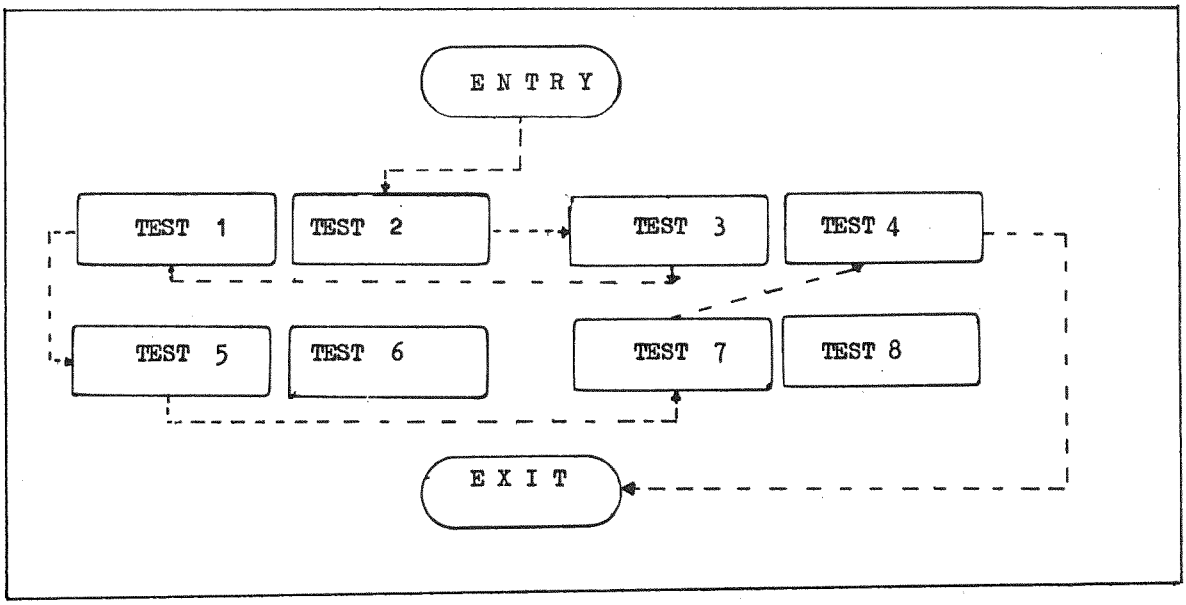
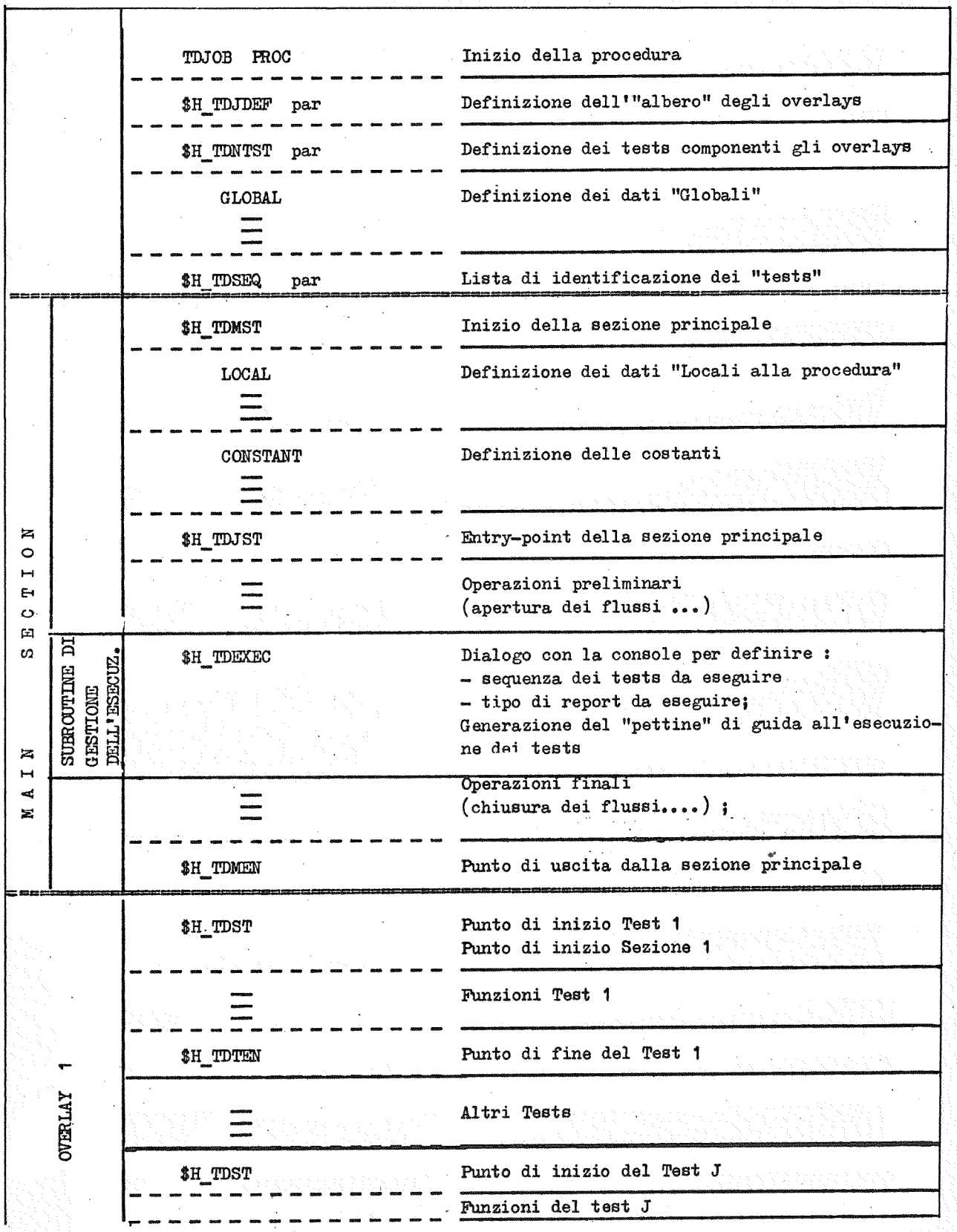


FIG. 4

5. UN ESEMPIO DI SCRITTURA DI UN PROGRAMMA DIAGNOSTICO

E' fornito nello schema di fig. 5.



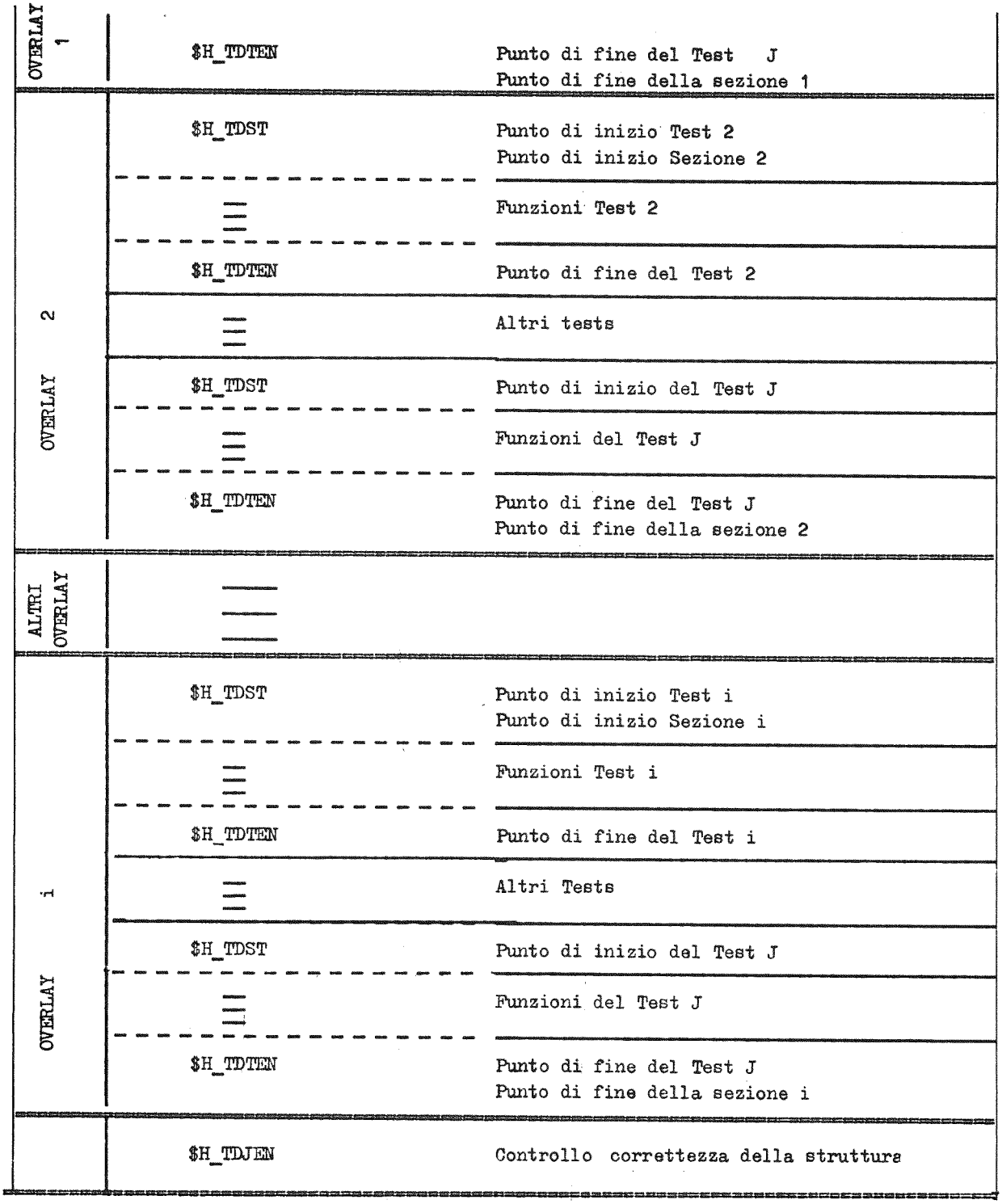

OVERLAY 1
$H_TDTEN
Punto di fine del Test J
Punto di fine della sezione 1
OVERLAY 2
$H_TDST
Punto di inizio Test 2
Punto di inizio Sezione 2
Funzioni Test 2
$H_TDTEN
Punto di fine del Test 2
Altri tests
$H_TDST
Punto di inizio del Test J
Funzioni del Test J
$H_TDTEN
Punto di fine del Test J
Punto di fine della sezione 2
ALTRI OVERLAY
OVERLAY i
$H_TDST
Punto di inizio Test i
Punto di inizio Sezione i
Funzioni Test i
$H_TDTEN
Punto di fine del Test i
Altri Tests
$H_TDST
Punto di inizio del Test J
Funzioni del Test J
$H_TDTEN
Punto di fine del Test J
Punto di fine della sezione i
$H_TDJEN
Controllo correttezza della struttura

FIG. 5

L ' USO DI MACRO J.C.L. PER IL TESTING PERSONALIZZATO

L. Trabattoni (°)

1. INTRODUZIONE

Nell'ambito dell'attività di sviluppo di sistemi si presenta il problema dell'esecuzione di insiemi di programmi di test rivolti alla verifica della correttezza di certi insiemi di funzionalità.

Alcune importanti caratteristiche che gli insiemi di test (nel seguito chiamati "catene") devono possedere sono:

La possibilità di essere eseguiti completamente o in parte, in dipendenza delle funzionalità da testare, in modo automatico cioè senza dover intervenire manualmente con operazioni di ricomposizione dei programmi.

La ripetibilità cioè l'esecuzione dei test non deve causare la distruzione o l'alterazione delle strutture d'interfaccia o di dati del sistema.

La possibilità di essere eseguiti sulle macchine disponibili al momento cioè essere indipendenti dalla configurazione di macchina.

La possibilità di essere eseguiti da persona senza particolari qualifiche specialistiche ad esempio dall'operatore addetto alla macchina. Dovrà essere il singolo programma di test a richiedere l'esecuzione delle manovre necessarie tramite colloquio con l'operatore via console.

Il controllo automatico dei risultati cioè la catena deve costruire come output un file di risultati che possa essere confrontato in modo automatico con un file campione in modo da riportare in visione al responsabile del testing solo le discrepanze verificatesi rispetto a quanto previsto dalle specifiche della catena.

L'uso di macro JCL può facilitare in modo notevole il raggiungimento di questi obiettivi.

(°) H.I.S.I. Pregnana Milanese.

2. L'USO DI MACRO JCL

Data una catena si può costruire una macro JCL per espandere un job-stream contenente le sequenze di chiamata di tutti i test della catena in modo che il processo di testing su un dato sistema e per un certo insieme di funzionalità si risolva con l'esecuzione di un job.

Poichè in generale l'esecuzione della catena non comporta l'esecuzione di tutti i programmi di test in essa contenuti, occorre disporre di variabili interne al macroprototipo che possano essere utilizzate come interfaccia tra la macro ed i programmi richiamati dalla macro. A questo scopo vengono utilizzate le Control-Variables (CV).

Una CV può essere locale al macroprototipo in cui è definita oppure globale cioè definita all'interno di un job ma esterna rispetto al macroprototipo in cui viene usata. L'inizializzazione può avvenire con valori costanti o con i valori dei parametri di chiamata della macro. Una CV può essere modificata all'esterno o all'interno di una macro con il verbo $LET oppure all'interno di un programma richiamato dalla macro con le primitive specifiche del linguaggio con cui il programma è scritto.

Nel macroprototipo la selezione dei cammini logici potrà avvenire in dipendenza dai valori dei parametri di chiamata e/o dai valori delle CV usando la frase $WHEN.

La personalizzazione della catena rispetto alla configurazione di macchina può essere ottenuta richiamando, come primo della catena, un programma di controllo dell'ambiente di sistema in cui la catena viene eseguita. Questo programma provvede ad analizzare i requisiti richiesti tramite i parametri di chiamata della macro ed in base a questi esegue dei controlli di compatibilità con la configurazione del sistema utilizzando le informazioni contenute nel supervisore residente. A seguito di questi controlli viene persona-

lizzato un insieme di CV che, una volta passate alla macro, verrà da questa utilizzato per la selezione dei programmi di test.

Il controllo automatico dei risultati può essere ottenuto facendo produrre ad ogni test un output su file; inoltre il test quando viene eseguito assegna ad una CV un valore prefissato in modo che sia possibile poi verificare se durante l'esecuzione della catena il programma è stato attivato.

L'ultimo programma della catena può essere dedicato all'analisi dei risultati dei test eseguiti; esso confronta i files di output dei programmi di test eseguiti con dei dati campione e produce un output a stampa mettendo in evidenza le differenze rispetto alle specifiche della catena.

3. ESEMPIO

Si voglia realizzare una macro JCL per il lancio di una catena di test sul software di gestione periferiche del livello 62.

La catena contiene un programma di test per ogni periferica prevista dal sistema; si vuole con un solo lancio della macro $TESTPERI, far eseguire i test corrispondenti a tutte le periferiche specificate nella chiamata della macro; se una periferica è collegata ma non è prevista dal System disk il test relativo non deve essere eseguito anche se richiesto nella chiamata della macro.

Una chiamata della macro potrebbe essere:

$TESTPERI DISK=MSU360,TAPE=MTU009,PRINTER=PRU004,MEDIA=TESTVOL,WORK=WDISK, ...

In figura è riportato un listing delle parti essenziali del macroprototipo della macro $TESTPERI. I programmi di test delle periferiche sono: TESTDISK, TESTTAPE, etc.
Le Control Variables EXEC1, EXEC2, etc. vengono utilizzate per registrare l'avvenuta esecuzione dei vari test e vengono lette dal programma COMPRES per il controllo dei risultati.

4. RIFERIMENTI BIBLIOGRAFICI

- GCOS lev.62 System Control. HISI Ord.No.AL11
- S.Farnedi,C.Poggiali,R.Polillo,A.Vignoni. TRP. Un sistema per la gestione ..... Atti del Congresso A.I.C.A. Milano 1976.

```
$TESTPERI DISK=MSU360,TAPE=MTU009,PRINTER=PRU004,...,MEDIA=TESTVOL,WORK=WDISK;
        $MAC ID=(DISK=NIL,TAPE=NIL,....);
        $DCV EXEC1,ITG=0;
        $DCV EXEC2,ITG=0;
        ....
        $LET CVDISK,'&DISK';
        $LET CVTAPE,'&TAPE';
        ....
        $STEP DEVCONTR;       :[CONTROLLO COMPATIBILITA' DELLA CONFIGURAZIONE]
        ...
        $ENDSTEP ;
        $WHEN @STATUS,=,0,JUMP=INIT;
        $SEND 'TESTPERI IMPOSSIBLE TO EXECUTE';
        $EXIT ;
INIT    $WHEN CVDISK,/=,NIL,JUMP=INITDISK;
        $WHEN CVTAPE,/=,NIL,JUMP=INITTAPE;
        $JUMP NOINIT;
INITDISK$PREPDISK ....;       :[INIZIALIZZAZIONE DISCO]
        $WHEN CVTAPE,/=,NIL,JUMP=NOINIT;
INITTAPE$PREPTAPE ....;       :[INIZIALIZZAZIONE NASTRO]
NOINIT  $CONTINUE ;
        $WHEN CVDISK,/=,NIL,JUMP=EXECDISK;
        $WHEN CVTAPE,/=,NIL,JUMP=EXECTAPE;
        ....
        ....
EXECDISK$ALLOCATE TESTDISKFILE1,MEDIA=&WORK,...  :[ALLOCAZIONE FILE RISULTATI]
        $ALLOCATE TESTDISKFILE2,MEDIA=&MEDIA,... :[ALLOCAZIONE WORKFILE]
        $STEP TESTDISK;
        $ASSIGN OUTFILE,TESTDISKFILE1,MEDIA=&WORK;
        $DEFINE OUTFILE,DVC=&CVDISK;
        $ASSIGN TESTFILE,TESTDISKFILE2,MEDIA=&MEDIA;
        $DEFINE ....
        $ENDSTEP ;
        $WHEN CVTAPE,=,NIL,JUMP=NOTAPE;
EXECTAPE$ALLOCATE TESTTAPEFILE1,....
        ....
        ....
ENDTEST $CONTINUE ;
        $STEP COMPRES ;       :[CONTROLLO RISULTATI DELLA CATENA]
        $ASSIGN FILE1,TESTDISKFILE1,...
        ....
        $ASSIGN OUTFILE,@OUT;
        $ENDSTEP ;
        $DEALLOCATE TESTDISKFILE1,MEDIA=&WORK;
        $DEALLOCATE TESTTAPEFILE1....
```

STRUTTURE DI DATI ASTRATTE IN ASSEMBLER MEDIANTE MACRO

C. Barbaglio (+)

1. INTRODUZIONE

Lo scopo di questo lavoro é quello di illustrare brevemente un insieme di macroprototipi che permettono la definizione di tipi di dati astratti nel linguaggio assembler del livello 62 (NAL).

A questo fine accenniamo brevemente al concetto di tipo di dati, di strutture di dati astratte e ai passi necessari per la definizione dei dati; mostriamo poi un esempio d'uso dei macroprototipi realizzati.

2. TIPO DI DATI [1]

Gli strumenti che abbiamo a disposizione per la descrizione del mondo reale sono spesso molto potenti (interi, reali, code, sequenze, coppie,...), mentre la rappresentazione all'interno di un calcolatore delle grandezze in gioco é sempre effettuata in termini di sequenze di bits.
Per ovviare a questo problema, i linguaggi di programmazione evoluti mettono a disposizione il concetto di tipo di dato. [1]

Possiamo definire il tipo di una variabile, costante o espressione, come un attributo che determina i valori che tale variabile, costante o espressione può assumere e gli operatori che su di essa possono essere utilizzati. (Così ad esempio in NAL una variabile di tipo H (halfword) può assumere valori interi da -32768 a 32767 e su di essa si può operare con istruzioni CG2, LG2, AG2, ...).

In un dato linguaggio di programmazione l'informazione sul tipo determina:

- La rappresentazione in memoria del valore della variabile (nell'esempio precedente binario in complemento a 2)
- L'ammontare di memoria che per tale variabile deve essere riservato (nell'esempio precedente 2 bytes)
- Il modo in cui devono essere interpretati gli operatori che ad essa vengono applicati.

Tipi semplici e tipi strutturati. E' possibile definire un tipo di dati in termini di altri tipi. Siamo così portati ad individuare:

- Tipi semplici o primitivi
- Tipi strutturati

I tipi semplici possono essere visti come attributi di oggetti che vengono definiti mediante assiomi (ad esempio, un intero si considera primitivo e non viene definito in termini di proprietà di altri tipi).
I tipi strutturati possono essere visti come tipi definiti, mediante opportuni "costruttori", in termini di altri tipi, tipi componenti, (primitivi o no) precedentemente definiti (ad esempio, si può pensare di avere il costruttore "array" che permette di ottenere, a partire dal tipo intero, reale, ecc. il nuovo tipo array di interi, reali, ecc.).

Quando dichiariamo una variabile di tipo intero, ci impegnamo ad assegnarle solo determinati valori e ad usare su di essa solo determinati operatori. Lo stesso vale per tipi strutturati: un array di interi deve essere riempito solo di interi, non più dalla sua lunghezza e su di esso si possono utilizzare solo determinati operatori.

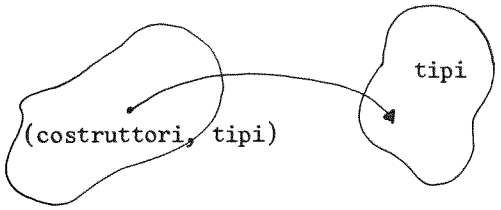
3. DEFINIZIONE DEI "DATI" DI UN PROGRAMMA

Il processo mentale che un programmatore esegue per la definizione dei dati di un programma é certamente assai complesso, e

(+) Istituto di Cibernetica -
Università degli Studi - Milano

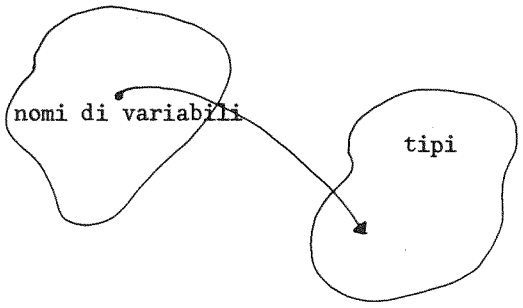
difficile da definire. Ad esempio, potremmo individuare le seguenti fasi:

- Dichiarazione dei tipielementari
  Innanzi tutto c'é la scelta di fondo di quali tipi considerare elementari. Di solito si sceglie, in modo esplicito, un sottoinsieme dei tipi primitivi del linguaggio.
  Risultato: una lista (eventualmente mentale) di nomi di tipi primitivi disponibili.
- Dichiarazione dei costruttori.
  Un discorso analogo al precedente vale per quanto riguarda la dichiarazione dei costruttori, cioé di quelle funzioni che permettono la costruzione di nuovi tipi mediante l'uso di quelli precedentemente definiti.
  Risultato: anche qui una lista con i nomi dei costruttori.
- Definizione dei tipi.
  Una volta a disposizione i tipi elementari e i costruttori si può pensare alla definizione dei nuovi tipi di dati che si vogliono utilizzare. (Questi tipi verranno definiti attraverso i costruttori e i tipi elementari precedentemente selezionati).
  Risultato: una lista di nomi dei nuovi tipi accompagnati dalla loro definizione:

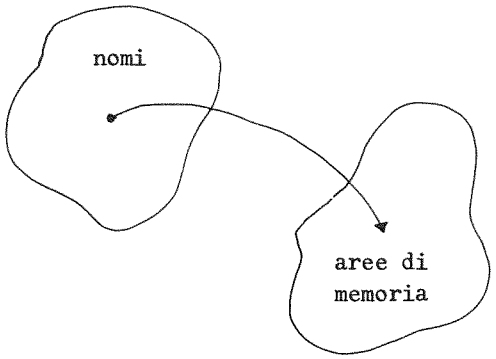


- Dichiarazione dei tipi delle variabili.
  A questo punto si può associare, ad ogni identificatore di variabile, il suo tipo.

  L'associazione può essere a questo livello puramente sintattica, cioé un nome di un tipo viene associato al nome di una variabile.
  Risultato: una tabella in cui all'identificatore della variabile é associato il nome del tipo:



- Allocazione.
  Ad ogni dato si deve far corrispondere l'area di memoria in cui tale dato deve essere allocato:
  Risultato: vengono scritte le istruzioni che eseguite rendono disponibile lo spazio di memoria necessario per inserire i dati:



4. STRUTTURE DI DATI ASTRATTE

Nel processo di programmazione, la definizione di strutture di dati non é un'attività che viene svolta in un preciso momento del lavoro, bensì essa é il risultato di una completa analisi del problema e di un insieme di decisioni che vengono prese durante lo sviluppo del programma.
Così, ad esempio, può capitare di sapere di dover accodare "richieste di servizi", e quindi di dover disporre di strutture a "code", senza però conoscere a priori tutti i dettagli delle informazioni da inserire in tali code.
Per questo potrebbe essere utile, in particolare per programmare in modo top-down,

avere a disposizione strumenti che permettano di definire strutture di dati e di descrivere le operazioni su di esse, ancora prima che tutti i tipi componenti siano completamente definiti.

Parleremo in questo caso di strutture di dati astratte. [2]

La possibilità di utilizzare tali tipi di strutture é fornita da alcuni linguaggi avanzati (ad esempio CLU), ma é utile poter utilizzare strutture di dati astratte anche in altri linguaggi, in quanto esse permettono:

- La realizzazione della programmazione strutturata non solo sui programmi ma anche sui dati.
- la definizione di tipi di dati adatti alla rappresentazione delle variabili in gioco nel problema.

Una delle caratteristiche fondamentali di tali strutture é quella che, con esse, é possibile scrivere porzioni di programmi definitive, che facciano riferimento a tipi di dati strutturati, per i quali non siano ancora stati specificati alcuni (o tutti) tipi componenti. Ad esempio, se A é un array costituito da elementi di tipo non ancora definito, si potranno eseguire operazioni del tipo B←A(I), senza sapere cosa sia l'oggetto A(I), ma, affinché questo sia lecito, é necessario che il trasferimento sopra scritto venga effettuato nello stesso modo indipendentemente dal tipo che verrà in seguito associato all'elemento A(I).

5. UNA REALIZZAZIONE IN NAL ATTRAVERSO L'USO DEL MACROPROCESSOR

Mostreremo ora un piccolo esempio con il quale vengono introdotte, in assembler NAL, strutture di dati astratte e vengono eseguite su tali strutture semplici operazioni.

Per realizzare questo esempio si é utilizzato il macroprocessor del livello 62. Si sono definiti dei macroprototipi che permettono di effettuare i vari passi descritti precedentemente per la definizione dei dati. Questi macroprototipi sono: $OBJCONST; $TYPE; $DCL; $CREATE;.

L'esempio che segue consiste nella allocazione di una variabile intera di uno stack di interi e di un vettore di stacks di interi.

Analizziamo ora brevemente il significato delle varie macro, mostrando la sequenza di chiamate necessarie per l'allocazione delle variabili sopra indicate.

$OBJCONST; permette di dichiarare i costruttori che si intenderanno disponibili, i cui nomi(INT,ARR1,STACK) verranno inseriti in una tabella di variabili di macroprocessor.

Nel nostro caso:

```
$OBJCONST    INT;
$OBJCONST    ARR1;
$OBJCONST    STACK;
```

risultato:

TABELLA DEI COSTRUTTORI	INT
	ARR1
	STACK

$TYPE;

permette di definire i tipi,elementari e non, che verranno utilizzati. Il primo parametro indica il nome del tipo, il secondo il costruttore utilizzato (nel caso di tipi strutturati), il terzo il tipo dei componenti. Anche qui le caratteristiche dei tipi verranno memorizzate in una tabella di macroprocessor.

Nel nostro caso:

```
$TYPE    INT,;
$TYPE    OBJ,STACK,OF=INT;
$TYPE    AROB,ARR1,OF=OBJ;
```

risultato:

TABELLA DEI TIPI	INT		
	OBJ	STACK	INT
	AROB	ARR1	OBJ

$DCL; Permette di associare ad ogni variabile un tipo. Il primo parametro indica il nome della variabile, il secondo il tipo. Ancora, ciò corrisponde alla compilazione di una tabella di macroprocessor.

Nel nostro caso:

```
$DCL    X,INT;
$DCL    Q,OBJ;
$DCL    P,AROB;
```

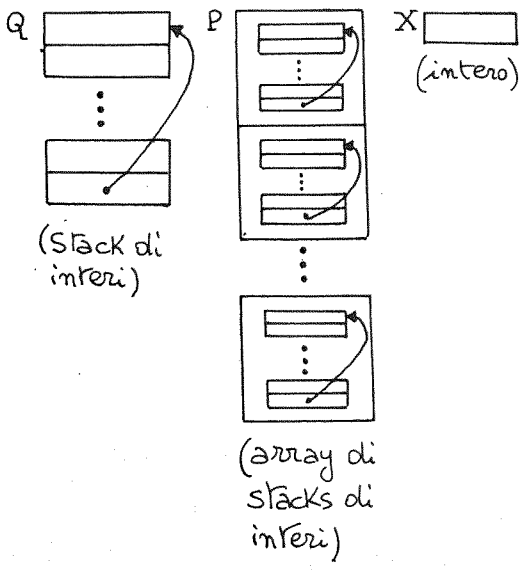
risultato:

TABELLA DEI SIMBOLI	X	INT
	Q	OBJ
	P	AROB

$CREATE; permette l'allocazione, richiamando i macroprototipi adatti, di oggetti il cui tipo sia stato precedentemente definito. Cerca nella tabella dei simboli il nome dell'oggetto da creare, gli associa il tipo e controlla nella tabella dei tipi che tutto sia a posto. Se ricerche e controlli vanno a buon fine, richiama il macroprototipo adatto per l'allocazione di oggetti del tipo associato all'oggetto da creare.
Nel nostro caso:

```
$CREATE     NAME=Q;
$CREATE     NAME=P;
$CREATE     NAME=X;
```

La prima CREATE, pertanto, assocerà all'identificatore Q il suo tipo (OBJ), e quindi a tale tipo la sua definizione. Nel nostro caso, OBJ é definito come il risultato dell'applicazione del costruttore STACK al tipo (elementare) INT. Per allocare la variabile Q, la macro CREATE non farà altro, in definitiva, che richiamare una macro di nome STACK la quale richiamerà, a sua volta, una macro di nome INT.
I macroprototipi INT e STACK, come tutte le macro associate ai tipi elementari e ai costuttori tipi, dovranno naturalmente essere forniti dall'utente.
Analogamente per l'allocazione di P e X.
In definitiva, le strutture allocate in memoria per Q, P, X saranno le seguenti:



L'accesso a tali strutture dovrà poi essere effettuato tramite opportune macro, anch'esse definite dall'utente, associate ai tipi elementari e ai costruttori.

Ad esempio una struttura definita mediante il costruttore di tipo STACK potrà essere operata mediante i macroprototipi

```
$PUSH
$POP
$TOP
```

mentre una struttura di tipo ARRAY (ad esempio definita mediante il costruttore ARR1) potrà essere operata mediante un macroprototipo
$MOVE elemento da inserire nell'array, nome array, posizione nell'array;
e simili. Un esempio più dettagliato di macroprototipi per l'accesso a strutture dati così definite é fornito in (3).

Così, mediante la macro-chiamata:

```
$PUSH     X,IN=Q;
```

viene introdotto l'oggetto X di tipo INT nello stack Q, mentre con:

```
$MOVE     NAM1=Q,NAM2=P,IDX2=2;
```

viene introdotto lo stack Q come secondo elemento dell'array P.
Si osservi come l'espressione delle $MOVE non dipenda dai tipi costituenti l'array P e lo stack Q: qualora tali costituenti cambiassero, non avremmo alcuna variazione in tale istruzione.

6. RIFERIMENTI

(1) C.A.R. Hoare, "Notes on Data Structuring", in Structured Programming, Academic Press, 1972
(2) B. Liskov, S. Zilles "Programming with abstract data types" Proceedings of ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices, V.9, n. 4, pagg. 50-59 (April 1974)
(3) F. Faidutti, "Un esempio di macro per gestire strutture di dati in assembler; il caso degli alberi", in questo numero di NOTE DI SOFTWARE.

UN ESEMPIO DI MACRO PER GESTIRE STRUTTURE DI DATI IN ASSEMBLER: IL CASO DEGLI ALBERI

F. Faidutti (+)

1. INTRODUZIONE

Le strutture di dati sono strumenti indispensabili alla programmazione. Alcuni linguaggi forniscono al programmatore strutture anche molto sofisticate (p.es. il Pascal), altri non ne forniscono: i linguaggi Assembler sono fra questi. Un programmatore Assembler deve quindi preoccuparsi di costruire le strutture di cui ha bisogno, usando i mezzi messi a disposizione dal linguaggio.

Questo articolo vuole essere un esempio di realizzazione di strutture di dati attraverso un macroprocessor. Le strutture di cui si parla sono gli alberi, il macroprocessor a cui ci si riferisce è quello del Livello 62 associato all'Assembler NAL.

Il discorso, in particolare, é inserito nell'ambito di strutture di dati astratte (1).

2. RICHIAMI SUGLI ALBERI

Un albero é una struttura di dati costituita da un insieme di nodi, connessi tra loro da rami. Ogni nodo non può avere più di un ramo entrante, mentre può avere più rami uscenti. Ogni nodo può avere un valore associato. Si può accedere ad un albero solamente attraverso la radice, la quale é l'unico nodo che non ha rami entranti.

Un esempio di struttura ad albero può essere il seguente:

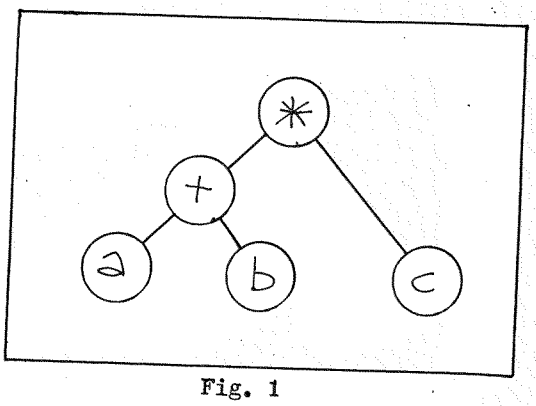


Fig. 1

Essa rappresenta, indicando anche l'ordine di esecuzione delle operazioni, l'espressione aritmetica (a + b) * c.

Strutture ad albero sono usate per analisi sintattiche, per ordinamento di numeri, ecc.

Una possibile scelta di primitive per gestire una struttura del genere é la seguente:

- una funzione di accesso alla struttura:
  - ..ENTER che permette di posizionarsi sulla radice
- due funzioni di percorrenza verticale:
  - ..DOWN per scendere di un livello a partire dal nodo attuale, (questa dovrà essere selettiva, per permettere di posizionarsi su uno tra i figli del nodo attuale)
  - ..UP per salire dal noto attuale al padre
- due funzioni di percorrenza orizzontale:
  - ..LEFT per spostarsi dal nodo attuale al fratello di sinistra
  - ..RIGHT per spostarsi dal nodo attuale al fratello di destra
- due funzioni per operare sull'albero:
  - ..INSERT per inserire una informazione nel nodo attuale
  - ..VISIT per estrarre l'informazione contenuta nel nodo attuale

(+) Istituto di Cibernetica - Università di Milano

. due funzioni per l'allocazione dinamica
..APPEND la quale permette di aggiungere un figlio al nodo attuale
..DELETE per deallocare il nodo attuale

3. UN SET DI MACRO PER LA GESTIONE DI ALBERI

E' stato realizzato (2) un insieme di macro per definire e gestire strutture ad albero binario nel linguaggio NAL, utilizzando il macroprocessor del Livello 62.

Poiché é desiderabile disporre di alberi i cui nodi contengano informazioni di tipo qualsiasi, questo é parametrizzato (p.es. il contenuto di un nodo di un albero potrebbe essere a sua volta una struttura di dati complessa (1)).

Macro per la definizione di un albero. Per definire un albero é necessario usare le macro di cui in (1). Un esempio è il seguente:

```
$TYPE TREEINT,TREE,OF = INTEGER;
$DCL ALBERO,TREEINT;
$CREATE NAME = ALBERO;
```

Con la prima macro-chiamata viene dichiarato il tipo 'TREEINT', a cui appartengono tutti gli alberi i cui nodi contengono valori interi; con la seconda si dichiara che la variabile ALBERO é di tipo 'TREEINT'. La chiamata della macro CREATE espande le dichiarative NAL per la definizione della struttura. In particolare espande:

. la radice, predisposta per contenere una informazione di tipo intero, con due puntatori 'liberi'
. un vettore i cui elementi sono nodi 'vituali': Ogni elemento cioé é provvisto di due puntatori liberi, e sarà agganciato all'albero con la funzione APPEND.

Macro che realizzano le funzioni di accesso. Le funzioni di accesso realizzate (cfr. § precedente) sono quelle illustrate in fig. 2.

In tutte, il parametro 'TREE' identifica l'albero su cui si vuol operare; nella DOWN il valore del parametro 'BRANCH' identifica uno dei figli (il valore '0' indica il primo da sinistra, '1' il secondo, ecc.); nella INSERT il valore di 'VALUEOF' specifica cosa inserire nel nodo, e in VISIT il valore di 'TO' specifica dove depositare l'informazione prelevata.

```
$ENTER TREE = <nome-albero> ;
$DOWN TREE = <nome-albero> ,
      BRANCH = <intero> ;
$UP TREE = <nome-albero> ;
$LEFT TREE = <nome-albero> ;
$RIGHT TREE = <nome-albero> ;
$INSERT TREE = <nome-albero> ;
        VALUEOF = <identifier> ;
$VISIT TREE = <nome-albero> ,
       TO = <identifier> ;
$APPEND { LEFT  } ,TREE = <nome-albero> ;
        { RIGHT }
$DELETE TREE = <nome-albero> ;
```

Fig. 2

4. UN ESEMPIO: UN PROGRAMMA DI ORDINAMENTO

Il programma seguente é un esempio d'uso della struttura esaminata. L'algoritmo é quello del 'tree-sort' (3), il quale ordina una sequenza di numeri in modo crescente, usando una struttura ad albero. Questa é uno strumento ausiliario: l'albero viene costruito in modo che ogni nodo abbia un contenuto maggiore del contenuto del figlio di sinistra e minore del contenuto del figlio di destra. Percorrendo opportunamente l'albero, si ha la sequenza di numeri ordinata. Si esce dalla struttura quando si incontra un nodo convenzionale (in figura, il nodo tratteggiato). (Fig.3).

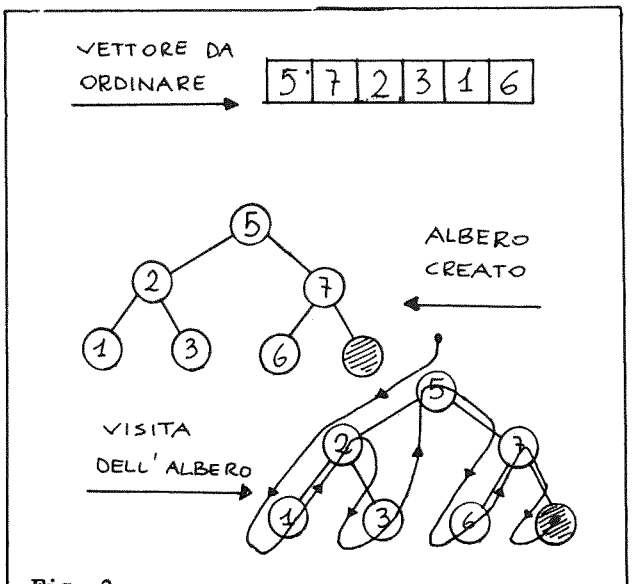


Fig. 3

Riportiamo il programma relativo scritto in STNal + macro per la gestione di alberi.

Il programma é diviso in tre fasi.

- creazione dell'albero : utilizza un file sequenziale in input (VETT) contenente numeri interi. Il primo elemento del file viene inserito nella radice. Si aggiungono poi nodi a destra o a sinistra, a seconda dell'esito del confronto con il nodo attuale, e vi si inseriscono ogni volta gli elementi. (Fig. 4)
- fase intermedia: per semplificare la condizione su cui terminerà l'algoritmo di visita, si aggiunge un nodo, contenente il valore esadecimale "FFFF", al nodo più a destra dell'albero. Se non è già presente, gli si aggiunge anche il nodo di sinistra. (Fig. 5)
- visita dell'albero: preleva le informazioni dall'albero, percorrendolo in modo opportuno, e le inserisce in un file sequenziale di output (ORDVETT). Ogni volta che si preleva una informazione si inseriscono nel nodo dei blanks per garantire una risalita corretta nel ciclo di UP.

Il programma é puramente esemplificativo. Sono stati omessi alcuni controlli necessari per una corretta esecuzione della terza parte del programma.

```
*CREAZIONE DELL'ALBERO
$H_OPEN VETT;
    $REPEAT;
        $H_GET VETT;
    $EXIT CMP = (CG2,'G4,CEOF',EQ);
        $ENTER TREE = ALBERO;
        $VISIT TREE = ALBERO,TO = RET;
        *All'ingresso della while, la prima volta, RET = BLANK.
        *Si esce quindi dal loop inserendo il primo elemento nella radice
        $WHILE CMP = (CCE,'RET,WA',GE),DO;
            *La seconda, e le volte successive, si aggiungono i nodi
            $SIF CMP = (CGIP,'GO,X"1"',EQ),THEN;
                $DOWN TREE = ALBERO BRANCH = 1;
                $SIF CMP = (CGIP,'GO,X"1"',EQ),THEN;
                    $APPEN RIGHT,TREE = ALBERO;
                $ENDIF;
            $ELSE;
                $DOWN TREE = ALBERO,BRANCH= 0;
                $SIF CMP = (CGIP,'GO,X"1"',EQ),THEN;
                    $APPEND LEFT,TREE = LABERO
                $ENDIF;
            $ENDIF;
            $VISIT TREE = ALBERO,TO = RET;
        $ENDDO;
        $INSERT TREE = ALBERO,VALUEOF = WA;
    $ENDREP;
$H_CLOSE VETT;
```

Fig. 4

NOTA: tutte le funzioni di accesso restituiscono un codice nel registro GO

In particolare GO contiene X"0" . se l'operazione é avvenuta correttamente

X"1" . se con una DOWN si arriva ad una foglia o
. se si tenta l'operazione a partire da una foglia

```
*FASE INTERMEDIA
$ENTER TREE = ALBERO;
*L'istruzione DOWN viene ripetuta fino a raggiungere l'ultimo
*nodo a destra
$WHILE CMP = (CGIP,'GO,X"1"',NE),DO;
    $DOWN TREE = ALBERO,BRANCH = 1;
$ENDDO;
$DOWN TREE = ALBERO,BRANCH = 0;
$SIF COMP = (CGIP,'GO,X"1"',EQ),THEN;
    $APPEND LEFT,TREE = ALBERO;
$ENDIF;
$UP TREE = ALBERO;
$APPEND RIGHT,TREE = ALBERO;
$INSERT TREE = ALBERO,VALUEOF = FFFF;
```

Fig. 5

```
*VISITA E ORDINAMENTO
$H_OPEN ORDVETT;
*L'istruzione DOWN viene ripetuta fino a raggiungere l'ultimo
*nodo a sinistra
$WHILE CMP = (CGIP,'GO,X"1"',NE),DO;
    $DOWN TREE = ALBERO,BRANCH = 0;
$ENDDO;
$VISIT TREE = ALBERO,TO = WA;
$REPEAT;
    $H_PUT ORDVETT;
    $INSERT TREE = ALBERO,VALUEOF = BLANK;
    *Si risale fino al primo nodo ancora contenente un numero
    $UNTIL CMP = (CCE,'WA,BLANK',NE)DO;
        $UP TREE = ALBERO;
        $VISIT TREE = ALBERO,TO = WA;
    $ENDDO;
    $H_PUT ORDVETT;
    $INSERT TREE = ALBERO,VALUEOF = BLANK;
    $DOWN TREE = ALBERO,BRANCH = 1;
    $VISIT TREE = ALBERO,TO = RET;
$EXIT CMP = (CCE,'RET,FFFF',EQ);
    *L'ultima down eseguita potrebbe aver inserito X"1" in GO.
    *E' necessario ripristinare il codice X"O" per poter eseguire
    *la DOWN che segue
    $WHILE CMP = (CGIP,'GO,X"1"',NE),DO;
        $DOWN TREE = ALBERO,BRANCH = 0;
    $ENDDO;
$ENDREP;
$H_CLOSE ORDVETT;
```

Fig. 6

RIFERIMENTI

(1) C. Barbaglio "Strutture di dati astratte in Assembler mediante Macro" - Note di Software - N. 2 - (1977)

(2) F. Faidutti "Macroprototipi per la realizzazione di strutture ad albero" - Rapporto interno - GEC - 1976

(3) D.E. Knuth "The art of computer programming" - Addison Wesley Publish.Co. (Massach.)

USO DI UN MACROGENERATORE NELLO SVILUPPO DI COMPILATORI FIRMWARE SPECIALIZZATI

F. Prigione(°)

1. INTRODUZIONE

Il Compilatore Firmware del calcolatore Honeywell Livello 62 realizza la traduzione dei micro-programmi che costituiscono il "firmware" dei diversi componenti del sistema.

Esso è stato disegnato per essere idoneo a risolvere il problema della traduzione di differenti linguaggi-sorgente in strutture-oggetto dipendenti dalle specifiche caratteristiche di diversi "packages" firmware.

Il suo attributo principale è pertanto una "alta adattabilità" ottenuta mediante la "parametrizzazione" della descrizione del linguaggio-sorgente e del linguaggio-oggetto in modo disaccoppiato. Tale "adattabilità" consente, facendo ricorso a macro, di sviluppare agevolmente delle versioni specializzate alla generazione di differenti linguaggi oggetto.

2. MOTIVAZIONI

Lo schema illustrativo del Compilatore Firmware appare in fig. 1.

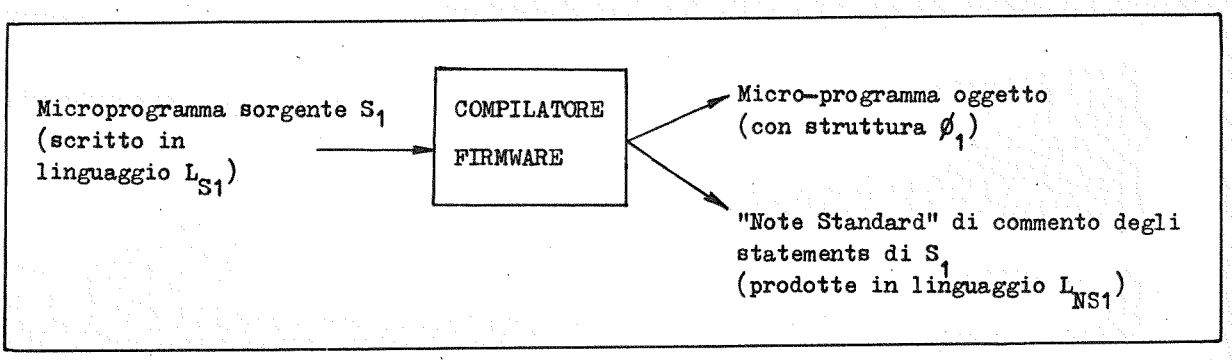


FIG. 1

Di ogni frase sorgente vengono realizzati due tipi di traduzioni, fra di loro indipendenti :

- la generazione della "stringa" - oggetto;
- la traduzione dell'espressione "sorgente", scarsamente auto-documentativa data la natura del linguaggio che prevede un numero elevato di argomenti per ogni verbo, in una espressione che la commenta ("Nota Standard").

Il requisito di "adattabilità" alla generazione di differenti strutture è soddisfatto realizzando lo schema di fig. 1 secondo la struttura illustrata in fig. 2.

(°) H.I.S.I. - Pregnana Milanese

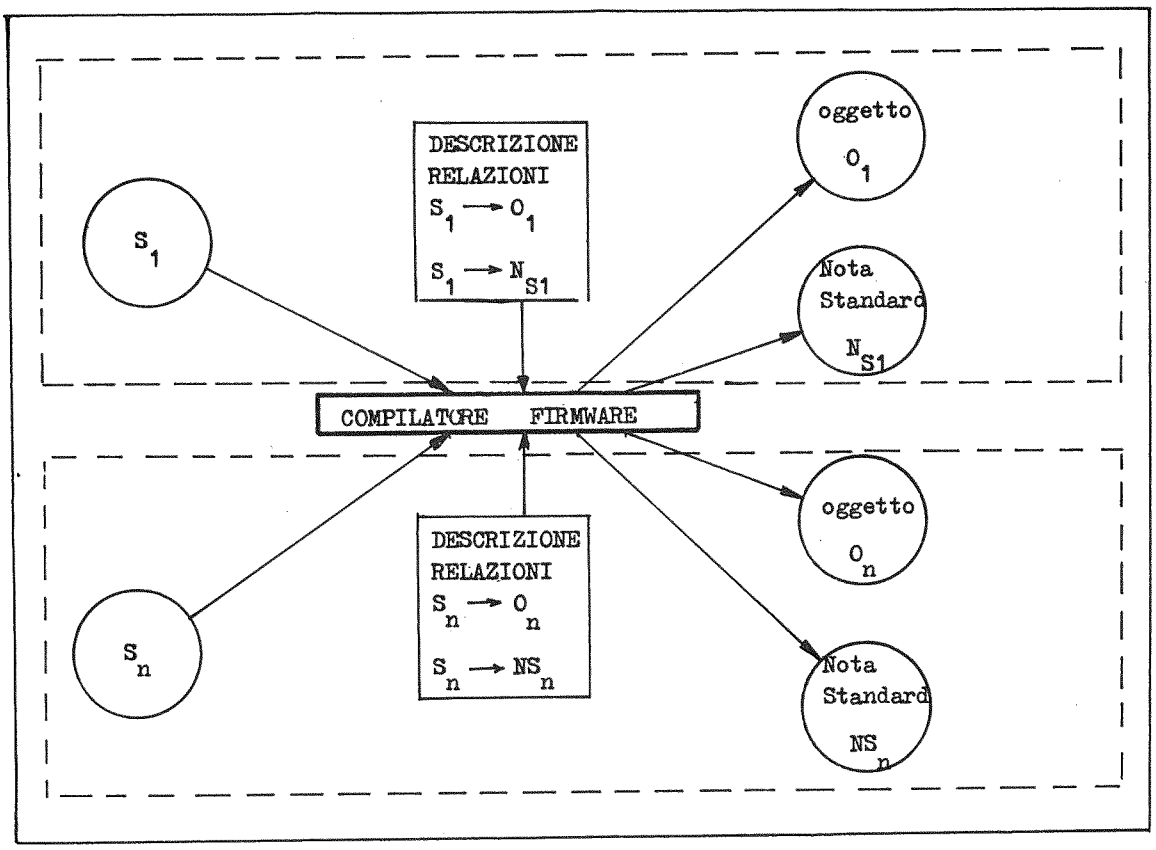


Fig. 2

3. STRUTTURA DEL COMPILATORE

L'"adattabilità" della struttura. Il Compilatore Firmware è costituito da un insieme di routines guidate da tavole di descrizione come rappresentato nella struttura di fig. 3.

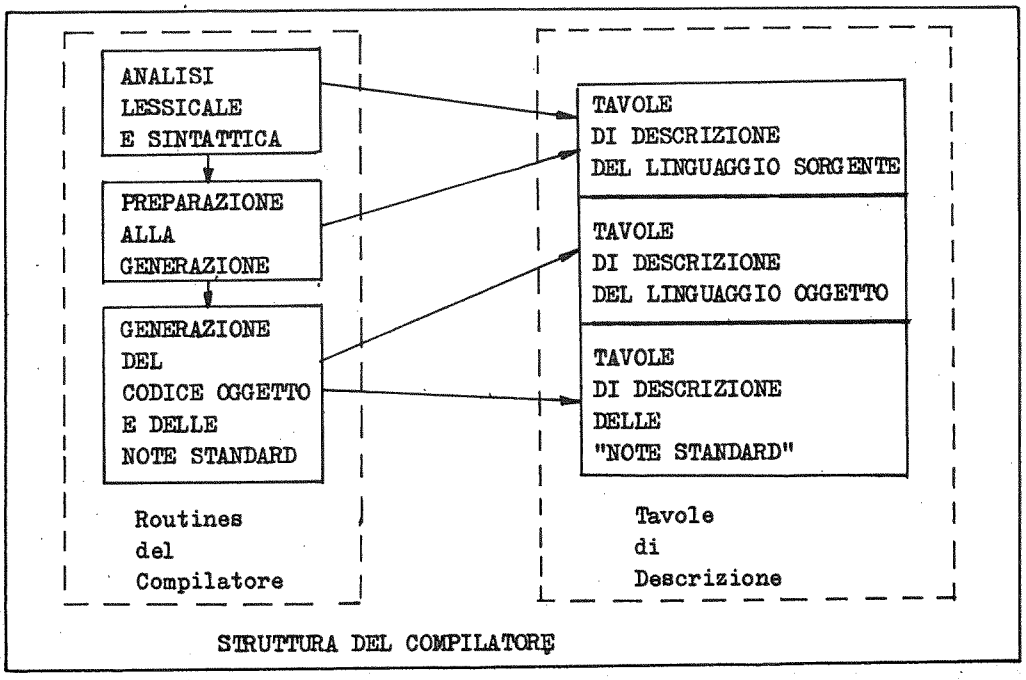


FIG. 3

La struttura descritta consente di produrre differenti versioni del Compilatore, in funzione unicamente dei contenuti delle tavole di descrizione, secondo lo schema di Fig. 4.

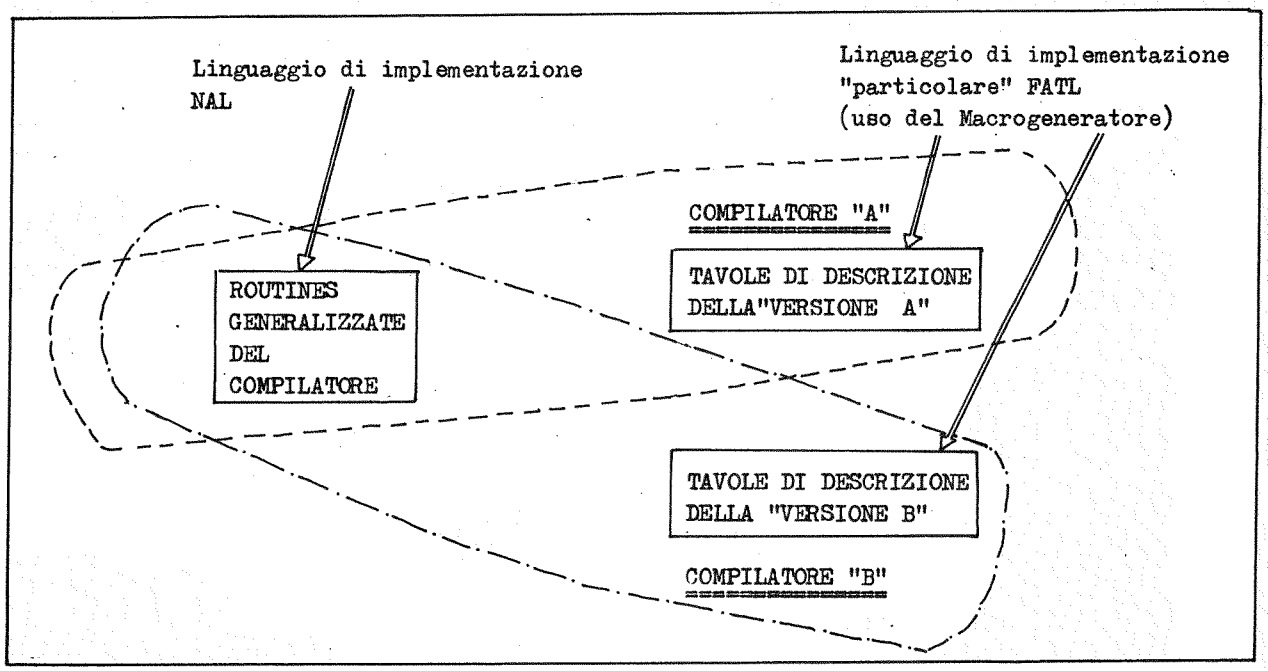


FIG. 4

I linguaggi coinvolti. La relazione fra i linguaggi coinvolti può essere espressa ricorrendo al diagramma a "T" di Fig. 5.

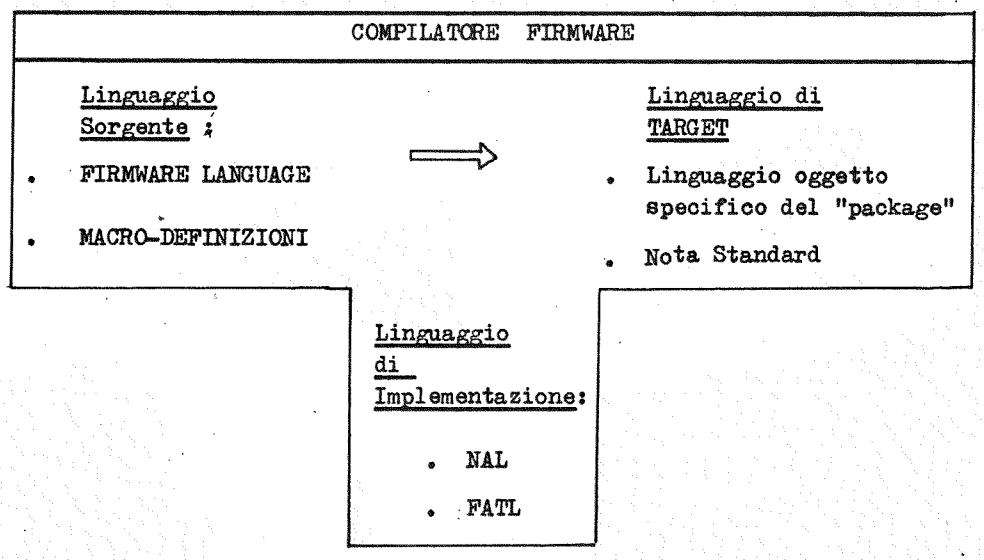


FIG. 5

Il linguaggio sorgente. E' generalmente definito in funzione della struttura oggetto associata al "micro-processor" fisico. Può essere qualsiasi, purchè appartenente a una "famiglia" (di tipo "Assembler") avente comuni regole grammaticali e sintattiche (risolte da algoritmi espressi nel corpo delle "routine generalizzate" del Compilatore).

L'espressione sorgente, a formato libero, è costituita da un "codice di funzione"avente per argomento una serie variabile di parametri (di tipo a parola-chiave, o auto - identificativi).

Ogni espressione sorgente è associata a una tavola di condizioni di compatibilità (sia "orizzontale" che "verticale") le quali, definendo le regole grammatiche e sin - tattiche, ne consentono il controllo e la diagnostica.

E' da rimarcare che il Compilatore Firmware consente di esprimere, all'interno del programma sorgente, chiamate di macro-definizioni interpretate sempre dal Macrogeneratore.

Questa funzionalità apre ai programmi firmware (micro-programmi) le stesse possibilità di uso del Macrogeneratore attualmente proprie dei programmi software.

Il linguaggio di implementazione. Il linguaggio di implementazione delle routines del Compilatore Firmware è l'Assembler Software (NAL) del livello 62.

La generazione delle tavole di descrizione è invece ottenuta mediante la definizione di un "particolare" linguaggio di implemen - tazione (Firmware Assembler Tables Language) ad alto livello e "orientato al problema". Tale linguaggio consente semplicità di scrittura nella descrizione dei linguaggi sorgenti e di target di ogni versione da sviluppare. Ed in virtù di questo fatto il Compilatore Firmware assume un requisito che rende alto il suo grado di "adattabilità".

Il "linguaggio di implementazione delle tavole" (FATL) è ottenuto facendo ricorso al Macrogeneratore e consente di generare in modo semplice e controllato le strutture di dati che compongono la descrizione di guida alla produzione del codice oggetto.

Il linguaggi di target. Sono due, e indipendenti.

Il primo è il "linguaggio oggetto" che definisce la composizione binaria della struttura oggetto.

Il secondo, cioè l'insieme delle "Note Standard", costituisce sostanzialmente la traduzione di una espressione "sorgente" in un'altra espressione, avente essa pure caratteristiche di linguaggio "sorgente".

L'USO DI MACRO PER LA GESTIONE SEMIAUTOMATICA DI CONDIZIONI ECCEZIONALI: L'ESEMPIO DI DACBOL

G. Zafferri (+)

1. INTRODUZIONE

Lo scopo di questa nota é di descrivere un insieme di macro per il trattamento delle condizioni di errore e/o eccezionali, che vengono segnalate dopo l'esecuzione di un verbo di data communications.

Il problema del trattamento degli errori, noto nella letteratura (1) col nome di "exception handling", può essere così schematizzato: un modulo ("operation's invoker") emette una richiesta di operazione, cioé richiama un altro modulo ("operation's executer") che esegue l'operazione stessa. Durante l'esecuzione dell'operazione vengono "sentite" delle condizioni anomale: per il trattamento di tali condizioni é necessario predisporre opportune procedure ("procedure di recovery") e inoltre stabilire quale modulo le debba attivare.

Nell'articolo vengono mostrate alcune possibili soluzioni del problema dello "exception handling". Viene poi esaminato il caso particolare di data communication e la soluzione di DACBOL. Infine viene mostrato un esempio di uso delle macro.

2. IL TRATTAMENTO DELLE CONDIZIONI ECCEZIONALI: ALCUNE SOLUZIONI

Affrontiamo qui il problema di chi debba trattare le condizioni rilevate.

La scelta più semplice é quella di riportare all'"operation's invoker" la segnalazione che una certa condizione si é verificata: l'"operation's executer" posiziona delle variabili di stato (o codici di ritorno) e le passa all'"operation's invoker" alla fine dell'operazione.
Questa scelta benché semplice e corretta (il modulo chiamante sa, in generale qual é la strategia di recovery opportuna, il modulo chiamato no) può essere svantaggiosa, soprattutto se il numero di segnalazioni da testare è elevato.

Un'altra possibilità è quella di spostare il trattamento delle condizioni occorse nel modulo che esegue l'operazione. Questo deve ricevere dall'"operation's invoker" indicazioni sulla strategia di recovery da seguire, ma in ogni caso é l'"operation executer" che effettua l'analisi delle condizioni rilevate e richiama le procedure opportune (le procedure stesse devono, in generale, però essere fornite da chi richiede l'esecuzione dell'operazione).

Questa scelta permette di centralizzare il trattamento delle condizioni eccezionali e di predisporre, almeno per alcune condizioni, dei trattamenti automatici: il modulo chiamato può, ad esempio, seguire una sua strategia di recovery se, ad esempio, non riceve indicazioni esplicite dal modulo chiamante.

3. UN CASO PARTICOLARE

Nella costruzione di programmi per data communication (°) ci si trova ad affrontare un tipico problema di "exception handling": il programma utente (programma applicativo) contiene delle primitive (*) SEND/RECEIVE attraverso cui esso può richiedere che

(+) Istituto di Cibernetica – Università di Milano

(°) in questo articolo si fa riferimento all'elaboratore HONEYWELL- Level 62

(*) il linguaggio di programmazione a cui ci si riferisce é il COBOL

venga effettuata un'operazione di trasmissione/ricezione. Il programma riceve, in due variabili di stato, dopo ognuno di questi verbi, indicazioni sulle condizioni eccezionali occorse durante l'esecuzione di una sua richiesta. Le segnalazioni passate all'utente sono un numero considerevole e sono di due tipi:

- segnalazioni di errore
  esempi: "errore durante la ricezione di un messaggio dal terminale xx", "errore durante la trasmissione di un messaggio al terminale xx", ...
- segnalazioni di eventi
  esempi: "tutti i terminali sono stati disabilitati", "il terminale xx é stato abilitato", ...

L'utente deve quindi (vedi anche figura 1):

a) riconoscere l'evento o l'errore che si é verificato, cioé eseguire un elevato numero di test sui valori delle variabili di stato
b) predisporre un trattamento per ogni errore o evento segnalato, cioé scrivere e attivare le procedure di recovery.

Il programma "avrebbe la vita facile" se "qualcuno" si preoccupasse di analizzare il contenuto delle variabili di stato e di richiamare le opportune procedure, in accordo alle sue richieste, lasciando a lui solo il compito di fornire le procedure stesse.

Questa richiesta può essere soddisfatta in modi diversi, più o meno sofisticati: una scelta semplice, che non comporta modifiche al software non applicativo, é quella di definire un insieme di macro che soddisfino i "desideri" del programma applicativo.

4. LA SOLUZIONE DI DACBOL

DACBOL é un insieme di macro (2):

$ENABLE... $SEND...
$CD.....
$DISABLE... $RECEIVER...

DACBOL é un insieme di nuove primitive per data communications che l'utente usa al posto dei verbi COBOL.
Esse sono di livello più alto delle primitive COBOL, nel senso che contengono al loro interno:

- il riempimento degli opportuni campi della CD (quando necessario)
- l'espansione del corrispondente verbo COBOL
- l'espansione di un modulo per l'analisi e il trattamento delle segnalazioni di errori e/o di eventi.

Queste primitive soddisfano le richieste viste al paragrafo 3 poiché:

- contengono la sequenza di test che identifica la particolare condizione verificatasi durante l'esecuzione di una richiesta di ricezione/trasmissione (risolvono quindi il problema a)
- permettono all'utente di specificare, attraverso opportuni parametri, le modalità di trattamento delle condizioni rilevate. L'utente può scegliere fra le seguenti possibilità:
  .. specificare quale procedura (scritta da lui stesso) debba essere richiamata dalla macro (con una frase PERFORM) quando si verifica una certa condizione o classe di condizioni
  .. specificare che per una certa condizione o classe di condizioni debba essere effettuato un trattamento standard (valore di default di alcuni parametri)

  (ciò risolve, per lo meno parzialmente il problema b).

5. UN ESEMPIO D'USO DI DACBOL

Nel paragrafo precedente abbiamo visto le caratteristiche generali di DACBOL.
In questo paragrafo non vogliamo darne una descrizione più dettagliata (per la quale si rimanda ai riferimenti citati), ma soltanto, con un esempio, sottolineare alcune caratteristiche.

Lo "scheletro" di un programma di data communications, ad esempio per un'applicazione di inquiry, che usa DACBOL é il seguente (vedi anche figura 2) (°)

(°) non sono indicati, per motivi di semplicità, tutti i parametri e quindi nemmeno tutte le procedure di recovery necessarie.

```
 01   AREA-MSG              PIC  X(100).
.........
 COMMUNICATION SECTION.
$CD ....
 PROCEDURE DIVISION.
    ......
$ENABLE ALL;
    ......
$RECEIVER INTO=AREA-MSG,-
          DATAMSG=MSG-PROC,THRU=END-MSG-
          PROC,ENABLE=ENABL-PROC,-
          INERROR=INPUT-ERROR-PROC,-
          ..........;
```

ENABL-PROC.

(procedura che invia al terminale, per cui é stata ricevuta la segnalazione che é stato abilitato, un messaggio di apertura)

INPUT-ERROR-PROC.

(procedura che effettua la disabilitazione del terminale su cui é stata sentita la segnalazione di errore)

MSG-PROC.

(procedura per il trattamento dei messagi arrivati dai terminali)

END-MSG-PROC.
EXIT

6. OSSERVAZIONI

- i trattamenti effettuati dalle procedure devono essere decisi dall'utente: i trattamenti mostrati nell'esempio sono solo indicativi
- la macro $RECEIVER, come si vede dalla figura, é un "loop" da cui (in condizioni normali) si esce quando viene raccolta la segnalazione "end of job" (parametro ENDJOB, non visibile nell'esempio)

  Mentre tutte le altre procedure vengono richiamate dalla macro $RECEIVER con una frase PERFORM, la procedura specificata nel parametro ENDJOB viene richiamata con un GO TO.
- La scelta di nascondere il ritorno in ricezione all'interno della macro é una scelta obbligata se si vogliono prevedere dei trattamenti automatici o semiautomatici di errore.

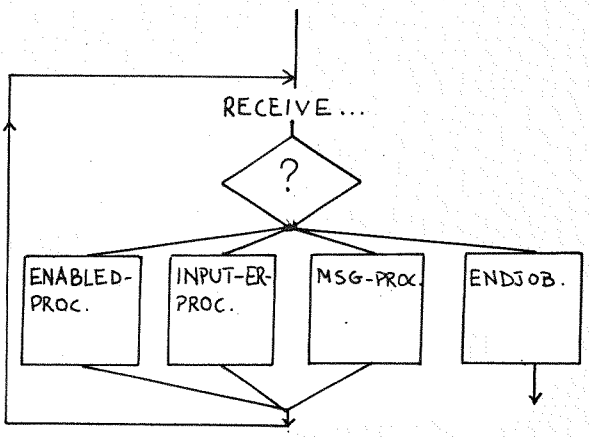


fig. 1

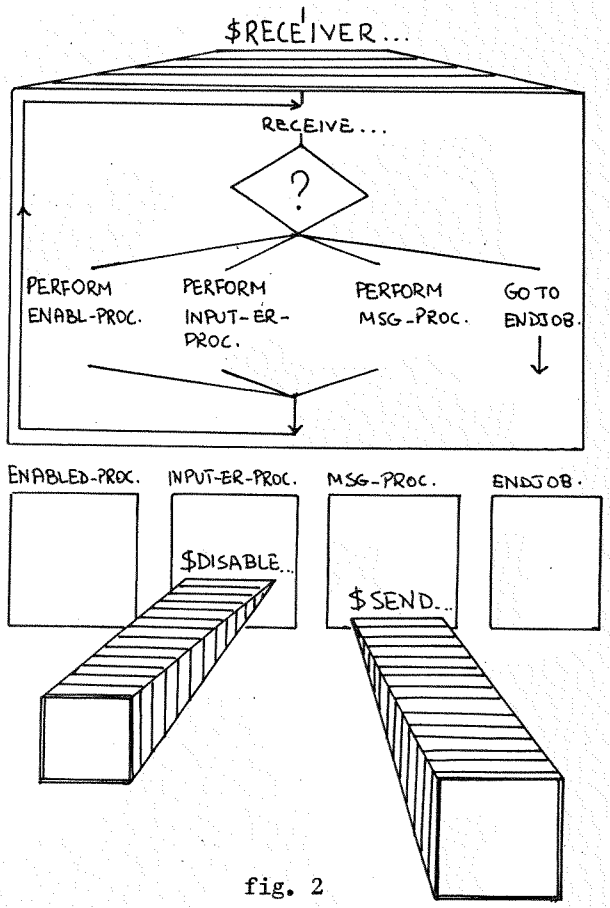


fig. 2

7. RIFERIMENTI

(1) Goodenough ,J.B., "Exception handling: issues and a proposed notation", Comm. ACM (dicembre 1975)

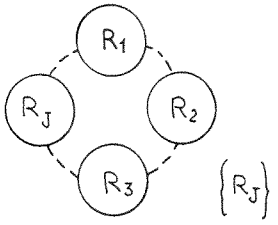
(2) Fouillouze,O., Polillo,R., Zafferri,G. "DACBOL: specifiche funzionali" - Istituto di Cibernetica, Università di Milano, Rapporto interno (maggio 1977)

STAND ALONE SYSTEM GENERATION LANGUAGE: LA GENERAZIONE DI UNA "IMMAGINE DI MEMORIA" ATTRAVERSO LA DEFINIZIONE DI MACRO

C. Monducci(°), M. Parini(°)

1. DEFINIZIONE DEL PROBLEMA

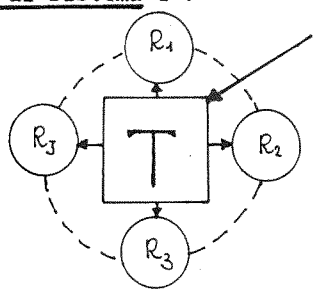
Tra gli scopi di questo linguaggio (1), fondamentale è quello di costituire un supporto all'emulatore della linea G 100 sul Livello 62. L'emulatore è un sistema completamente realizzato a firmware (2) che realizza un "adattamento" tra l'hardware del sistema emulante e il e il software del sistema emulato, caricando, interpretando ed eseguendo sul L 62 istruzioni, e quindi programmi, scritti per la linea G 100. Il sistema è un insieme di moduli (routines firmware) ciascuno dedicato all'espletamento di una specifica funzione.



Si distinguono :

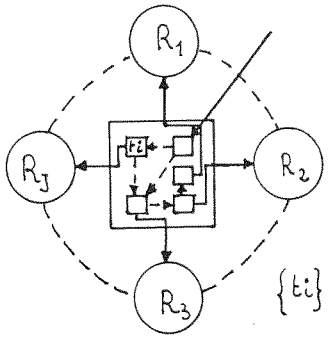
- routines di sviluppo delle istruzioni G 100
- routines di gestione dei devices
- routines che realizzano funzioni di sistema (interpretazione istruzioni, memory-dump, inizializzazione, ecc.).

Tutte queste routines sono collegate da una tavola di sistema T :



T istituisce le vie di comunicazione fra le funzioni elementari del sistema e ne realizza le interfacce. La tabella T è a sua volta composta in "tabelle elementari" $\{t_i\}$ fra loro interconnesse.

I collegamenti sono realizzati in due modi. Vi è un collegamento esplicito, realizzato tramite descrittore (pointer); in questo caso la tabella da raggiungere è puntata da un elemento descrittore appartenente ad una tabella già visitata. E' implicito il collegamento a tavole allocate in posizioni fisse di memoria. Le routines funzionali sono collegate in modo esplicito.



L'insieme delle tabelle elementari, le informazioni in esse contenute e le routines "puntate" sono funzioni della configurazione emulata/emulante e costituiscono la "immagine di memoria" che deve essere generata.

E' quindi necessario per ogni possibile combinazione di configurazioni emulata/emulante :

- descrivere le configurazioni
- selezionare un particolare sottoinsieme di tabelle elementari appartenenti a $\{t_i\}$ che definiscano la tabella di sistema T
- realizzare la rete di puntatori tra queste tabelle
- individuare un particolare sottoinsieme di routines appartenenti a $\{R_j\}$ necessario per

(°) H.I.S.I. - Pregnana Milanese

l'emulazione
- realizzare i collegamenti (puntatori) tra le tabelle e le routines.

Ad esempio per emulare una unità nastro di G 100 con un'unità nastro (pluridevice) di L 62 è indispensabile, tra l'altro, disporre dei seguenti elementi :

- numero del canale fisico del device emulante (PCCT)
- numero del canale logico del device emulante (LCCT)
- stato della porta del device emulante e indicazione della routine di gestione del device (PCT)
- entry alla routine di gestione del device (R)
- stato logico del device emulato (LCT)
- associazione device emulato/emulante (CONFIGURATION TABLE) cioè disporre in memoria tra l'altro, di una struttura del tipo :

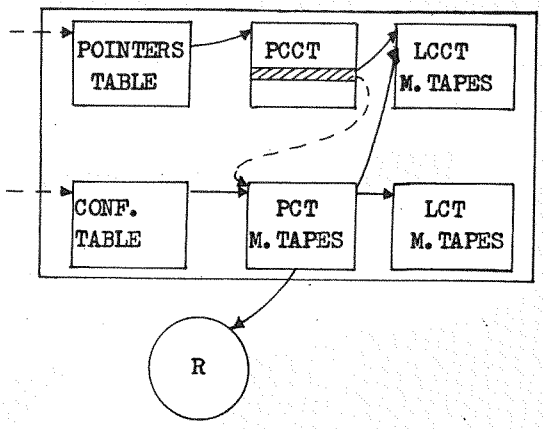
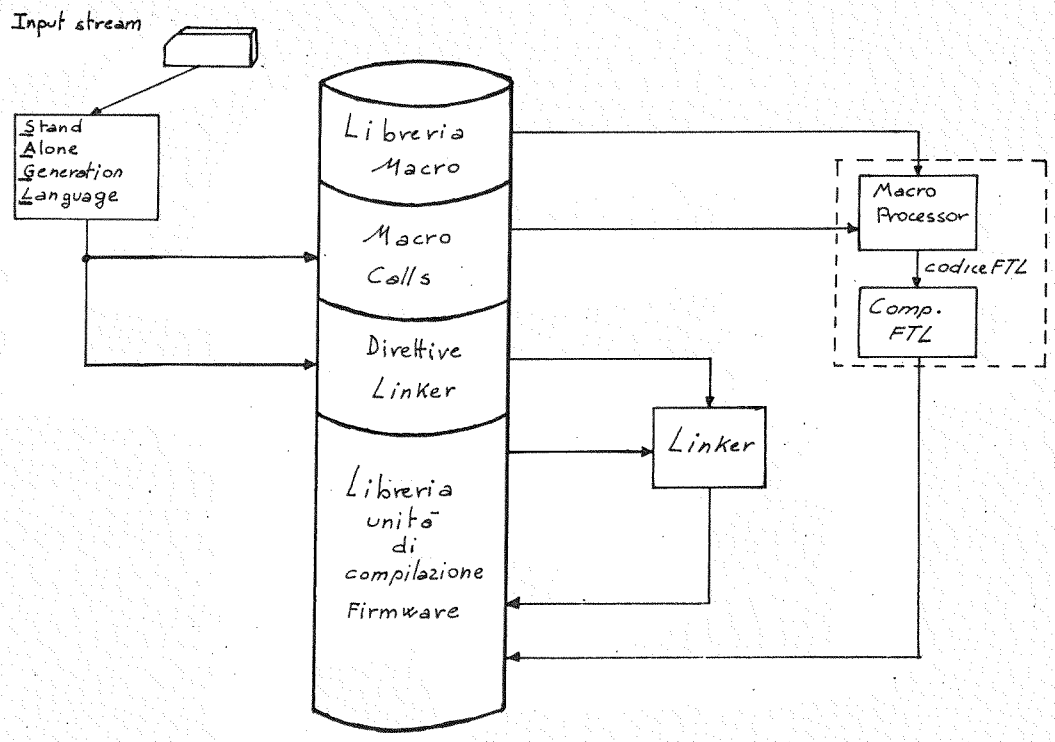


FIG. 1

dove i collegamenti espliciti sono rappresentati da linee continue, quelli impliciti da linee tratteggiate.

2. SOLUZIONE SCELTA

E' schematizzata nel seguente flow-chart:



Descrizione delle configurazioni. E' fatta su schede, con apposito linguaggio (SAGL), e il controllo e l'analisi di queste genera :

- macro calls parametriche rispetto ad un insieme di parametri $\{P_i\}$ che contiene le informazioni caratterizzanti le configurazioni
- direttive al linker

Generazione delle tabelle. E' la fase fondamentale ed è realizzata con delle macro, una per ogni possibile formato delle tabelle elementari (PCCT, PCC, LCCT, LCT, ecc.).

Queste macro sono parametriche rispetto a particolari sottoinsiemi di $\{P_i\}$ ed espandono codice FTL.

FTL è un linguaggio firmware per la definizione di tabelle che permette di definire costanti ed indirizzi in forma simbolica (il relativo compilatore FTL associa ai nomi simbolici gli indirizzi delle relative labels).

Ogni macro è quindi in grado di generare una classe di tabelle di formato t_i, essendo la particolare tabella attualizzata dal valore dei parametri.

Le decisioni sono prese con strutture di controllo del tipo IF.....THEN (3) che operano sui parametri e su dati comuni di tipo GLOBAL.

Riprendendo l'esempio citato (fig. 1), che vuole comunque essere solo indicativo della metodologia adottata, per realizzare le tabelle descritte in figura è, tra l'altro, necessario generare le seguenti macro :

```
$PHISCONF (...MT=YES,...);
$POINTERS (...);
$PCT      (...DEV=CLUSTER,LCNUMB=6,...);
$PCCT     (MPA=NO,PCNUMB=2,MONODEV=NO,
          TYPE=MT);
$LCCT;
$LCT;
```

dove i parametri attualizzati indicano che nella configurazione emulata non esistono multiple port adapter (MPA=NO), e che una unità nastri della linea G 100 deve essere emulata (MT=YES) con un'unità CLUSTER del L62 (DEV=CLUSTER,TYPE=MT),pluridevice (MONODEV=NO), provvista di sei canali logici (LCNUMB=6) e collegata sul canale numero 2 della configurazione emulante (PCNUMB=2).

A fronte di queste chiamate il Macro Processor espande il codice FTL che, una volta compilato, nella fase successiva, definisce in modo completo le tabelle.

Puntatori tra le tabelle. Vengono realizzati dal COMPILATORE FTL che opera sul codice generato dalle macro e produce una unità di compilazione firmware nella quale tutti i collegamenti interni alle tabelle sono attualizzati. E' questa l'immagine della tavola T sulla quale debbono ancora essere realizzati i collegamenti con le routines funzionali.

Routines. Il sistema di generazione riceve in ingresso le routines distribuite in modo già compilato. I riferimenti dalle tabelle alle routines sono risolti dal LINKER (firmware) che opera sulla tavola di sistema, ormai definita, e sulle routines presenti in libreria. E' necessario rispettare delle convenzioni nella scelta dei nomi dei possibili entry-points delle routines in modo da semplificare la generazione delle direttive al linker alla semplice indicazione delle routines da linkare.

La scelta delle routines non richiede informazioni aggiuntive a quelle analizzate per la creazione delle tabelle.

3. CONCLUSIONI

Il processo qui delineato, per le sue caratteristiche di generalità, può essere utilizzato per risolvere a basso costo i problemi della generazione della "immagine di memoria" del sistema firmware, del complesso cioè delle strutture logiche che costituiscono l'ambiente nel quale il firmware opera.

In più il sistema consente una rappresentazione formale sufficientemente chiara degli elementi di tali strutture.

Bibliografia :

1) C.Monducci: FW Factory nativa, rapporto interno Honeywell

2) A.Brioschi: Introduzione al firmware. Quaderni di Informatica HISI N.2,Agosto 74

3) Macro Processor Complex FS,MN93.

LA COSTRUZIONE DI MACROPROTOTIPI STRUTTURATI.

M.Maiocchi - R.Polillo
Ist.Cibernetica-Milano

RIASSUNTO

Vengono fornite alcune indicazioni di carattere pratico per la costruzione "strutturata" di macroprototipi, nel linguaggio del macroprocessor del Livello 62.

1. INTRODUZIONE

Un macroprototipo è un programma. Come tale esso riceve dati in ingresso, effettua elaborazioni, produce risultati.

La programmazione strutturata ha fornito negli ultimi anni indicazioni sul modo di costruire i programmi in modo che essi risultino leggibili e modificabili, ed ha fornito, in taluni casi, anche indicazioni metodologiche per determinarne la struttura.

E' questo il caso di metodologie come quella di Jackson (2) e di Warnier (1) che, occupandosi di classi di problemi molto specifiche (cioé programmi EDP in cui le operazioni da effettuare sono molto semplici, ma si ha una grande quantità di dati in ingresso e in uscita da gestire) forniscono indicazioni molto precise su come procedere per determinare la struttura del programma desiderato.

Tali metodi permettono, partendo dall'analisi dei dati che il programma deve manipolare, di determinare la struttura del programma stesso in modo tale che essa risulti il più possibile "simile" a quella dei dati da elaborare. Così, a sequenze di dati dello stesso tipo che si ripetono corrisponderanno, nel programma, strutture iterative; a dati di tipo diverso che si presentano in alternativa corrisponderanno, nel programma, strutture di selezione.

Metodologie di questo tipo pongono l'accento sulle strutture dei dati che si presentano in ingresso al programma da costruire: si potrebbe infatti pensare di modellare il programma sulla struttura che i dati in uscita devono avere, ma poiché l'ordine con cui i dati da elaborare si presentano in ingresso é spesso un dato di fatto non alterabile, risulta conveniente che questi siano gli elementi per determinare la struttura del programma.

Così le metodologie suddette descrivono i flussi sequenziali di dati in ingresso come alternative tra dati differenti, ripetizioni di dati simili, o sequenze di dati. Ad esempio, la "carta sintattica" di fig.1 (5) rappresenta, in termini di sequenza, di selezione e di iterazione un file sequenziale di ingresso composto di records di tipo a,b,c,p,q,m,n disposti nel seguente ordine: prima un certo numero di ripetizioni della terna a,b,c, poi un numero di ripetizioni della coppia p,q, infine un certo numero di ripetizioni di un record che può essere o m o n.

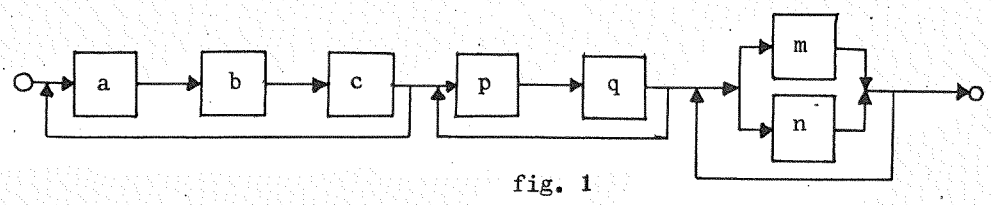


fig. 1

Conseguentemente il programma per elaborare tale file si presenterà costituito di tre blocchi in sequenza: il primo costituito da una iterazione all'interno della quale verranno elaborati in sequenza i records a,b e c; il secondo costituito da un'iterazione nella quale verranno elaborati in sequenza i records p e q; il terzo costituito da una iterazione nella quale verranno elaborati in alternativa i records m e n.

2. IL CASO DEI MACROPROTOTIPI

Nel caso dei macroprototipi ci troviamo di fronte ad una situazione abbastanza diversa: mentre i risultati in uscita si presentano generalmente come stringhe di caratteri prodotte in modo sequenziale, i dati in ingresso sono forniti esclusivamente come parametri (o come valore di variabili globali di macroprocessor), e quindi "in parallelo", senza la possibilità di individuare alcuna struttura.

Ciò suggerisce di applicare, per la progettazione di macroprototipi, criteri analoghi a quelli descritti per i programmi, che però si basino esclusivamente sulla struttura dei risultati da produrre. In tal modo, non solo si avrà a disposizione una metodologia per la costruzione di macroprototipi, ma non si avranno neppure quei problemi che si incontrano, nel caso di programmi EDP, ogniqualvolta ci siano profonde differenze fra la struttura dei dati in ingresso e la struttura dei dati in uscita ("structure clashes", (2)).

3. UN ESEMPIO

Per comprendere l'uso del metodo, supponiamo di dover costruire una coppia di macro per la programmazione strutturata, per la realizzazione di strutture di iterazione enumerativa (ci appoggiamo qui al linguaggio del macroprocessor (3) del Livello 62 ed al suo linguaggio assembler, il Nal (4).

La sintassi delle macro sia la seguente:

```
$DO VARYING = nome-registro,
     FROM = {valore intero | identificatore},
     TO   = {valore intero | identificatore},
    {INCR | DECR} = {valore intero | identificatore};
.....
$ENDDO;
```

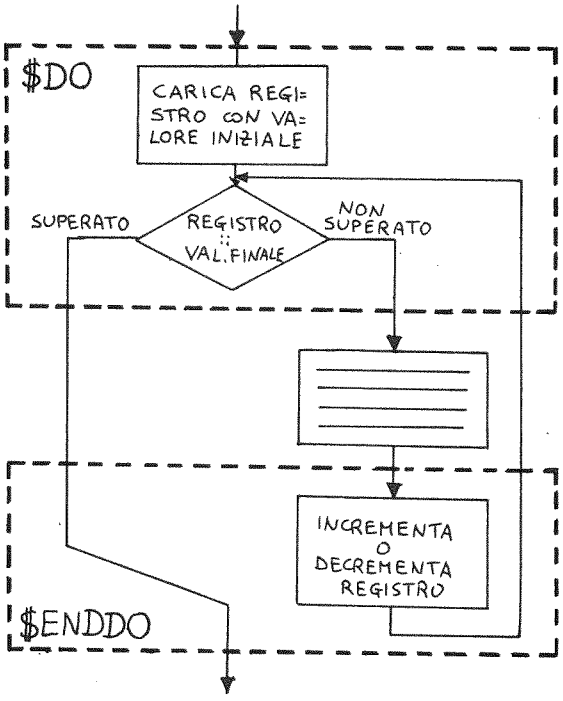


fig. 2

Il codice espanso dalle macro dovrà ricalcare il modello mostrato in fig. 2, tuttavia il tipo specifico delle istruzioni che devono essere generate dipenderà sia dai parametri presenti (INCR o DECR), sia dai tipi dei dati (valori numerici piccoli usabili come immediati, valori numerici più grandi, nomi di variabili di memoria). La struttura del codice da generare può essere allora descritta visivamente dalle due carte sintattiche di fig. 3.

Avendo definito in tal modo la sintassi della sequenza di frasi da generare, la struttura del macroprototipo è completamente determinata (fig.4).(Si osservi che, poichè nella espansione della $ENDDO sono necessarie alcune informazioni note solo al momento della chiamata della $DO, in questa sono state posizionate delle opportune variabili globali. Si noti, inoltre, che la sintassi del linguaggio del macroprocessor è stata semplificata per una migliore leggibilità). Analogamente si procederà per la macro $ENDDO. Si osservi come sia stato osservato uno standard di codifica che prevede sempre la traduzione delle strutture con un solo punto di ingresso ed un solo punto d'uscita. I vari controlli sulla natura dei valori dei parametri di chiamata, nel prototipo appena citato, potranno essere realizzati,

ad esempio, con macro di livello inferiore che forniscano come valore di ritorno l'esito del controllo: si sarà così costruito il prototipo in modo completamente top down.

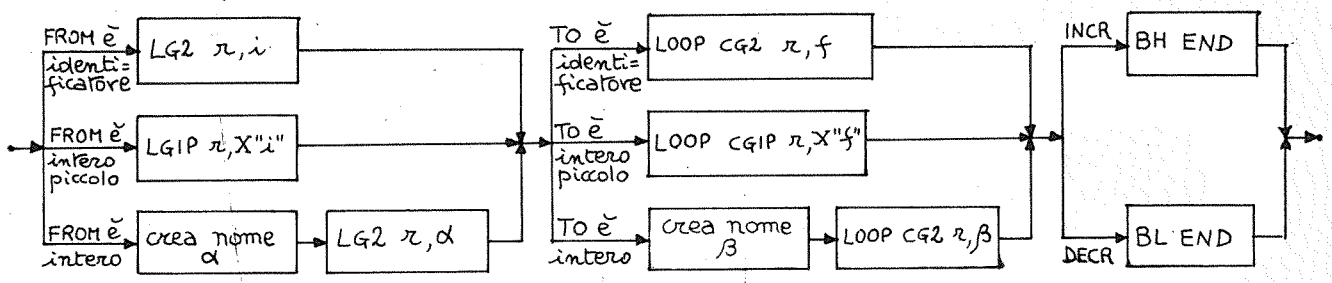
Se nell'esempio si é fatto uso di sole strutture di selezione e di sequenza, possiamo vedere il seguente, in cui si usano anche strutture di iterazione. Supponiamo di dover realizzare una macro per l'allocazione di tabelle, ad esempio:

$TABLE name,LELEM=lungh.1-elemento-di-tabella, NELEM=n.-elementi-della-tabella;

Il codice espanso dipende chiaramente dai valori dei parametri: se il numero di elementi di tabella richiesti non supera 256, é possibile usare una sola dichiarativa DC Nal con un fattore di ripetizione opportuno, altrimenti é necessario espandere una successione di DC fino al completa-

---

$DO VARYING = r, FROM = i, TO = f, {INCR / DECR} = p;



$ENDDO;

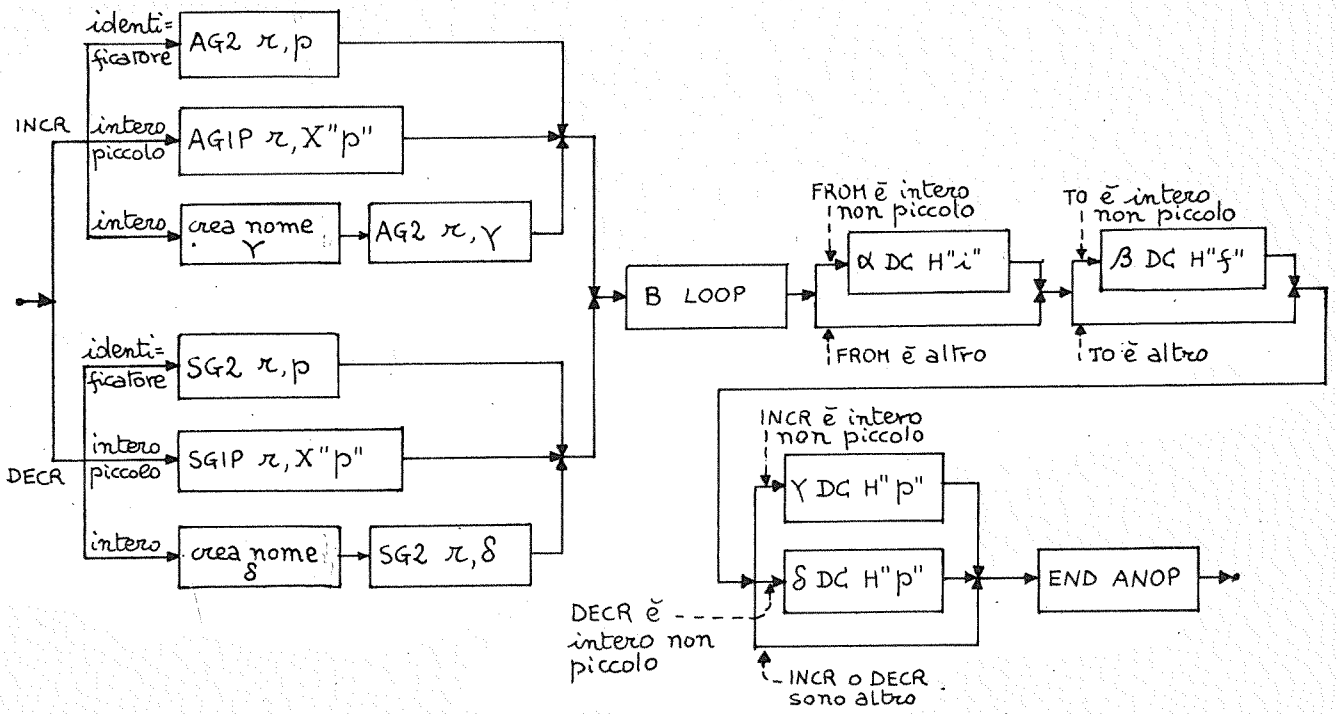


fig. 3

```
$DEF DO (↑VARYING,FROM,TO,INCR,DECR);
    $IF %FROM è identificatore     THEN
                                   $GOTO NM;;
    $IF %FROM è un numero piccolo THEN
                                   $GOTO NP;;
          crea un nome
          LG2   %VARYING,nome←
          $GOTO EIF;
    $NM:
          LG2   %VARYING,%FROM←
          $GOTO EIF;
    $NP:
          LGIP  %VARYING,X"%FROM"↵
    $EIF:

    $IF %TO è identificatore       THEN
                                   $GOTO TNM;;
    $IF %TO è un numero piccolo    THEN
                                   $GOTO TNP;;
          crea un nome
          LOOP  CG2   %VARYING,nome←
          $GOTO EIF2;
    $TNM:
          LOOP  CG2   %VARYING,%TO←
          $GOTO EIF2;
    $TNP:
          LOOP  CGIP  %VARYING,X"%TO"↵
    $EIF2:

    $IF %INCR = NIL THEN $GOTO DECR;;
          BH    END↵
          $GOTO EIF3;
    $DECR:
          BL    END↵
    $EIF3:

          memorizza in variabili
          globali tipo e valore dei
          parametri, per la espan-
          sione di $ENDDO
$END;
```

fig. 4

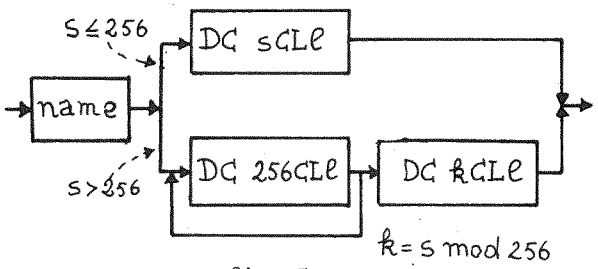


fig. 5

```
$DEF TABLE (NAME ↑ LELEM,NELEM);
    $LOCAL SZ = %NELEM;
    %NAME
    $IF %NELEM > 256 THEN $GOTO GT;;
          DC    %NELEM CL%LELEM←
          $GOTO FIN;
    $GT:
    $LOOP:
          DC    256 CL%LELEM←
          $SET SZ = SZ-256;
          $IF SZ > 256 THEN $GOTO LOOP;;
          DC    $VALUE SZ;CL%LELEM↵
    $FIN:
$END;
```

fig. 6

mento della tabella. Il codice espanso avrà quindi la struttura rappresentata dalla carta sintattica di fig. 5.

Il macroprototipo avrà allora la struttura mostrata in fig. 6.
In esso il locale SZ é stato usato per il calcolo di k=s mod 256. Anche qui si é usata una forma di codifica per la struttura iterativa che mantiene un solo punto d'ingresso e di uscita.

4. STRUTTURA GENERALE DI UN MACROPROTOTIPO

Negli esempi precedenti si é completamente trascurato il controllo della correttezza dei parametri d'ingresso. Ovviamente tale operazione deve essere fatta, ma non presenta particolari problemi.
Occorre tuttavia procedere in modo, per quanto é possibile, standardizzato. Ad

esempio si potrà procedere nel seguente modo:

- assegnare a tutti i parametri un valore di difetto, uguale a NIL, se il problema non lo richiede;
- verificare l'esistenza di ciascun parametro non opzionale;
- verificare la compatibilità tra i parametri opzionali che devono essere contemporaneamente presenti o in alternativa;
- verificare la correttezza del valore di ciascuno dei parametri esistenti;
- posizionare, per ciascuno di questi controlli, un indicatore che permetta di ricordare la rilevazione di un errore nella chiamata;
- in caso di errore emettere gli opportuni messaggi e non procedere all'espansione del codice;
- in caso di assenza d'errori, espandere il codice richiesto.

Lo schema generale di un macroprototipo risulta quindi quello di fig. 7.

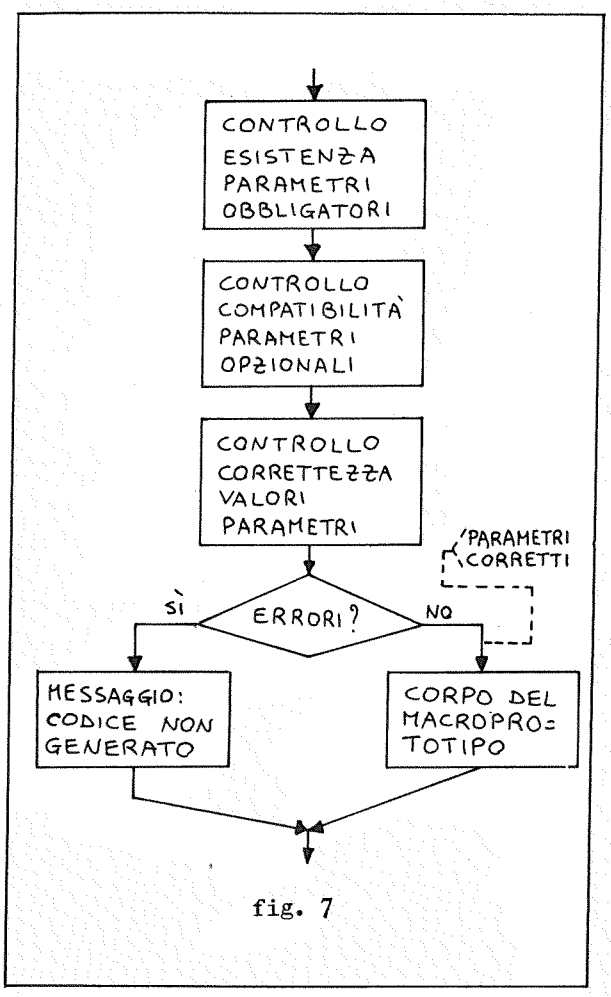


fig. 7

5. RIFERIMENTI

(1) J.D. Warnier, B.M. Flanagan, "Entrainement à la programmation", Les éditions d'organisation - Paris

(2) M.A. Jackson, "Principles of Program Design", Academic Press, 1975

(3) Macro Processor FS-HISI

(4) A. Savani "Student's guide to P6 Programming", Documento HISI - 1972

(5) C.A.R. Hoare and N. Wirth "An Axiomatic definition of the Programming Language PASCAL" , International Summer School on structured programming and programmed structures - Munich 1973.

AVVENIMENTI

HONEYWELL SOFTWARE PRODUCTIVITY SYMPOSIUM

1. INTRODUZIONE

Il 26, 27, 28 Aprile 1977 si è tenuto a Minneapolis un simposio, organizzato dalla Honeywell, sulla produttività del Software.

Il simposio aveva lo scopo di favorire un dialogo aperto tra le differenti compagnie associate alla HIS sulle metodologie e sugli strumenti utilizzati e sperimentati, con l'obiettivo di rinforzare la produttività nel processo di produzione del Software.

Il simposio, il primo nel suo genere, è stato accolto con molto entusiasmo da tutti i partecipanti che, molto numerosi, sono pervenuti dalle varie organizzazioni HIS e Consociate di tutte le parti del mondo.

2. STRUTTURA DELLA CONFERENZA ED OSSERVAZIONI GENERALI

Durante il simposio, sono stati presentati più di 30 "papers" organizzati in 10 sessioni, riguardanti i seguenti argomenti:

- Metodologie di programmazione;
- Linguaggi e preprocessatori;
- Metodologie di gestione e strumenti;
- Metodologie di prova;
- Studi e realizzazioni di particolare interesse.

Le diverse sessioni hanno rivelato che approcci simili, anche se applicati a problemi diversi, sono stati utilizzati da gruppi differenti, e che molti strumenti e metodologie potrebbero essere trasferiti.

Il problema base della produttività, invece, è stato trattato più esaurientemente nelle varie discussioni aperte e nel "panel" finale dove sono state presentate le seguenti tematiche:

- Interpretazione comune della produttività come linee di codice generate che portano e delle errate conclusioni;
- Metodologie di gestione che possono influenzare moltissimo la produttività del Software, con la necessità, quindi, di programmare le esperienze ed i risultati per valutarne i vantaggi;
- Principali difficoltà che si presentano nell'aumentare l'efficienza e che si trovano negli aspetti educativi, ossia la diffusione organizzata delle metodologie e l'insegnamento degli strumenti.

3. ARGOMENTI TRATTATI

Metodologie di Programmazione

In questa sessione, sono state presentate le indicazioni e gli standard-guida per l'uso di tecniche di implementazione strutturata.

Di particolare interesse, è stata la presentazione di Chittenden, il quale ha utilizzato il metodo degli "automata" a stati finiti per descrivere un programma nello sviluppo di una parte di un sistema "communication".

Da segnalare anche il lavoro di Boyd, il quale ha presentato una metodologia di progetto chiamata "WELLMADE", che aveva molti punti in comune con lavori di Dijkstra e McGee - Pizzarello.

E' stato anche mostrato un interesse generale nello sviluppo "TOP-DOWN" ed anche nella dimostrazione della correttezza dei programmi, nella specificazione dei linguaggi e nella adeguatezza dei linguaggi di programmazione alle tecniche strutturate.

Linguaggi di programmazione e preprocessatori

La sessione era orientata ai linguaggi da adoperare per implementazione di sistemi operativi, compilatori ecc., con particolare enfasi sulla definizione della struttura dei dati, sulla separazione degli aspetti semantici da quelli sintattici e sulla modularità. Nel campo dei preprocessatori ne sono stati presentati diversi a supporto della programmazione strutturata. Particolarmente interessante è stato il lavoro di Shinozawa il quale ha presentato un preprocessatore uguale sia per il Cobol che per il Fortran trasportabile su tre livelli di sistema operativo ACOS 2-4-6 (corrispondenti al GCOS 62-64 - 66).

Metodologie di gestione e strumenti

In questa sessione è stato segnalato l'impatto dell'uso della programmazione strutturata in un ambiente di ampia produzione ed è stata analizzata la struttura e l'ambiente di produzione. La sessione è stata di alto interesse ed in particolare sono da segnalare i lavori di Mc Gee-Pizzarello, quello di Negre sulla produzione del GCOS 64, e quello di Plouviez Cohen. Quest'ultimo ha presentato un sistema di Data Base integrato per tenere sotto controllo il processo completo di produzione del software.

Metodologie di prova

In questa sessione si è parlato di strumenti, metodi ed esperienze con evidenziazione del controllo di qualità inteso sia come verifica funzionale del prodotto che come una attività completamente integrata con l'implementazione.

Studi e realizzazioni di particolare interesse

Sono state presentate molte esperienze fatte con la programmazione strutturata e nel la costruzione di strumenti. Da notare in particolare l'implementazione di un "FLAGGER" sulla linea L 6.

Il Laboratorio (HISI) di Pregnana ha partecipato con i seguenti lavori:

- C. Montibelli, J. Bristow - "Tools to Support Structured programming and to Produce Automatic Design Documentation".
- A. Brioschi, F.Gariboldi, M.Maiocchi - "A consistent set of Tools for Testing and Structured Programming".
- A. Cazziol, C. Cucciati, G.Guella - "Evaluation of Testing Efforts as a way to Productivity in Software Development"
- L. Antonelli, M. Barbato - "Experience of a System Software Module Implementation Using Structured Programming and Hardware Interface Simulation".

(M. BARBATO)

BIBLIOGRAFIA

INTRODUZIONI AI MACROGENERATORI

P.J. Brown, MACRO PROCESSORS AND TECHNIQUES FOR PORTABLE SOFTWARE, John Wiley and Sons, London - New York - Sydney - Toronto, 1974, pagg. 244.
Un libro di facile lettura, che costituisce una introduzione abbastanza completa ai macroprocessors e al loro uso per la costruzione di software portabile.

"The book is split up into three parts. Part 1 covers macro processors. The aim is not to survey every macro processor in existence, but to present a reasoned exposition of the subject, extracting the most interesting and useful concepts and developing these to the full.

Part 2 covers software portability. The relationship of this to Part 1 is that one of the most important uses of macro processors is to aid software portability, and, conversely, a good proportion of portability techniques centre on the use of macro processors. It would certainly be shortsighted, however, to think that all techniques for software portability should use a macro processor, and Part 2 covers several techniques that do not. The primary aim of Part 2 is to bring all the techniques together and to evaluate their relative merits...

Pratical experience is one of the best ways of learning, and the purpose of Part 3 is to give the reader a chance to do something for himself. It describes, in full detail, how a specific piece of portable software can be implemented on virtually any computer..."

Indice:
Part 1 - MACRO PROCESSORS (Basic concepts, Special purpose and general purpose, the IBM OS Macro Assembler, The PL/I Macro Processor, GPM and the TRAC language, ML/I and STAGE2, Communication between macro, Implementation, Uses and limitations, Compiler-integrated macros.
Part 2 - SOFTWARE PORTABILITY (Use of macros for software writing, Some practitioners of DLIMPs (Descriptive Language Implemented by Macro Processor), Making Compilers portable, Families of descriptive languages.
Part 3 - A PORTABILITY PROJECT (Description of the project, The ALGEBRA System, The kernel of LOWL, Mapping and documentation, The LOWN kernel test program, Listing of LOWLTEST, MD-logic and LOWL extension for ALGEBRA, Listing of ALGEBRA, Testing and implementation of Algebra, Writing Software il LOWL.

M. Campbell-Kelly, AN INTRODUCTION TO MACROS, Macdonald/American Elsevier Computer Monographs, pagg. 122 (1973).

Un volumetto di lettura assai semplice.
"The book begins with a discussion of the macrofacilities found in macro-assemblers, and goes on to describe refinements and applications of macros in this context.
There follows an examination of some general purpose macroprocessors and their application to portable programming. Finally, the role of macros in high livel languages is considered".

N. Solntseff, A. Yezerski, A SURVEY OF EXTENSIBLE PROGRAMMING LANGUAGES, in Annual Review in Automatic Programming, 7 (M.I. Halpern, W.C.McGee eds), Pergamon Press (1974), pagg. 267-307.
Una interessante rassegna che colloca i macro generatori nel più ampio ambito dei linguaggi estensibili: "A review of various approaches to the design and implementation of extensible programming languages is presented. The languages are classified according to the stage of the translation process during which the extension mechanism operates. The characteristics and the requirements of extension mechanisms are discussed".

IL SEGNALIBRO

In questa rubrica vengono segnalati libri, articoli o studi di recente pubblicazione che, a giudizio della redazione o dei lettori di NOTE DI SOFTWARE, rivestono un particolare interesse nell'ambito dei temi a cui questa rivista é dedicata.

Le citazioni fra virgolette, se non ne é indicata la fonte, sono tratte dai lavori stessi.

Linguaggi di programmazione: Modula, Euclid, COL, Algol 68

N. Wirth, MODULA: A LANGUAGE FOR MODULAR MULTIPROGRAMMING, Software-Practice and Experience, Vol. 7, 3-35 (gennaio-febbraio 1977).

"This paper defines a language called Modula, which is intended primarily for programming dedicated computer systems, including process control systems on smaller machines. The language is largely based on Pascal, but in addition to conventional block structure it introduces a so-called module structure. A module is a set of procedures, data types and variables, where the programmer has precise control over the names that are imported from and exported to the environment. Modula includes general multiprocessing facilities, namely processes, interface modules and signals. It also allows the specification of facilities that represent a computer's specific peripheral devices. Those given in this paper pertain to the PDP-11."

N. Wirth, THE USE OF MODULA, Software-Practice and Experience, Vol. 7, 37-65 (gennaio-febbraio 1977).

"Three sample programs are developed and explained with the purpose of demonstrating the use of the programming language Modula. The examples concentrate on the uses of modules, concurrent processes and synchronizing signals. In particular, they all focus on the problems of operating peripheral devices. The concurrency of their driver processes has to occur in real time. The devices include a typewriter, a card reader, a line printer, a disk, a terminal with tape cassettes and a graphical display unit. The three programs are listed in full".

N. Wirth, DESIGN AND IMPLEMENTATION OF MODULA, ibidem, 67-84

"This paper gives an account of some design decisions made during the development of the programming language Modula. It explains the essential characteristics of its implementation on the PDP-11 computer, in particular its run-time administration of processes and the mechanism of signalling. The paper ends with some comments on the suitability of the PDP-11 for this high-level multiprogramming language."

G.J. Popek, J.J.Horning, B.W. Lampson, J.G. Mitchell , R.L. London, NOTES ON THE DESIGN OF EUCLID, Proceedings of an ACM Conference on Language Design for Reliable Software, Sigplan Notices, Vol. 12, n.3, (Marzo 1977)

"Euclid is a language for writing system programs that are to be verified. We believe that verification and reliability are closely related because if it is hard to reason about programs using a language feature, it will be difficult to write programs that use it properly. This paper, discusses a number of issues in the design of Euclid, including such topics as the scope of names, aliasing, modules, type-checking, and the confinement of machine dependencies; it gives some of the reasons for our expectation that programming in Euclid will be more reliable (and will produce more reliable programs) than programming in Pascal, on which Euclid is based."

B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, G.L. Popek, REPORT ON THE PROGRAMMING LANGUAGE EUCLID, Sigplan Notices, Vol. 12, n. 2. (febbraio 1977).

Contiene la descrizione estesa del linguaggio.

A. Evans Jr, C.R. Morgan, DEVELOPMENT OF A COMMUNICATIONS ORIENTED LANGUAGE, Bolt Beranek and Newman, Inc., U.S. Department of Commerce, NTIS AD-A027 523 (Marzo 1976). COL, "a high order language designed to be maximally effective as a vehicle with which to program communications applications ..."

A. van Wijngaarden et al., REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 68, Sigplan Notices, vol.12, n. 5, maggio 1977. Contiene la descrizione completa del linguaggio.

P.G. Hibbard, A SUBLANGUAGE OF ALGOL 68, ibidem.

Fortran

FOR-WORD, FORTRAN DEVELOPMENT NEWSLETTER, (pubblicato dall'Ad Hoc Committee on Fortran Development, ACM-Sigplan) Sigplan Notices, Vol. 12, N.4 (Aprile 1977) Contiene informazioni sullo stato del nuovo Fortran 1977.

Multiprogrammazione

H. Wettstein, THE IMPLEMENTATION OF SYNCHRONIZING OPERATIONS IN VARIOUS ENVIRONMENTS, Software-Practice and Experience, Vol. 7, 115-126 (gennaio-febbraio 1977).

"Since E.W.Dijkstra's work on the mutual interdependencies of co-operating sequential processes there has been much discussion on various synchronizing mechanisms. This discussion usually leads to abstract definitions of synchronizing primitives. However, their realization proceeds differently in various environments. Length of critical regions, number of processors, and dispatching strategy have to be taken into account. In this paper is presented a family of synchronizing operations, the members of which are all based on the same fundamentals. They have been designed with the concept of structured programming in mind."

Linguaggi di disegno e documentazione

R. F. Rosin, A GRAPHICAL NOTATION FOR DESCRIBING SYSTEM IMPLEMENTATION, Software-Practice and Experience, Vol. 7, 239-250 (Marzo-Aprile 1977).

"It is suggested that many important concepts in computer system implementation are more difficult to describe and understand than would be the case were there a suitable graphical notation available. Such a notation, which consists of five elements and two forms for composing these elements, is introduced. It is then used to illustrate some basic techniques in the implementation of programming languages". La notazione é particolarmente adatta a descrivere sistemi in cui compaiono interpreti, emulatori, simulatori, "as well as multilayered software and hardware".

E. Denert, SPECIFICATION AND DESIGN OF DIALOGUE SYSTEMS WITH STATE DIAGRAM, International Computing Symposium, Liege, 4-7 aprile 1977, pgg. 417-424 Un semplice linguaggio adatto alla specificazione top-down di dialoghi (ad es., realizzati da sistemi di tipo transazionale).

Un nuovo libro di Dijkstra

E.W.Dijkstra , A DISCIPLINE OF PROGRAMMING Prentice-Hall (1976)

Non é un libro per inesperti. Non é nemmeno un manuale che descrive un linguaggio o una metodologia. E' un libro che propone alcune tesi sulla programmazione e una metodologia (una "disciplina scientifica") per la costruzione di programmi corretti, attraverso la discussione di un vasto numero di esempi di diversa complessità e natura. Il cuore della tesi sostenuta é che la costruzione di un programma e la dimostrazione della sua correttezza devono essere momenti di uno stesso processo che si sviluppa partendo dalla chiara definizione del problema in una asserzione finale e costruendo il programma come un trasformatore di tale asserzione nelle condizioni iniziali. Per rendere naturale questa sintesi del programma dall'asserzione finale Dijkstra usa delle semplici istruzioni basate sui "guarded commands" che non impongono al programmatore la definizione di sequenze ordinate di istruzioni, ma semplicemente la identificazione delle condizioni possibili e delle istruzioni da eseguire per ciascuna delle condizioni. Il libro, di poco più di 200 pagine, si snoda in 28 brevi capitoli così organizzati: 0-6, Introduzione e Elementi basici della Teoria e del Linguaggio di Programmazione; 7-8, Applicazioni a problemi elementari;9-12, Altro sulla Teoria, Dichiarazioni dei dati (tipi e strutture); 13-23, Applicazioni, problemi più sostanziosi (Sort, Merge, Patter Match, Update di un file sequenziale);

24-25, Applicazioni a problemi matematici complessi (Calcolo di una superficie tridimensionale convessa); 26-27, Riflessioni e conclusioni. Non é una lettura semplice, ma é certamente una lettura proficua, soprattutto se si segue il consiglio dell'autore di provare a risolvere i problemi proposti prima di leggerne le soluzioni nel libro. (G. De Michelis).

Calcolatori a stack.

Il numero di maggio (vol. 10, n.5, 1977) di Computer contiene una serie di articoli sulle "stack machines": D.M. Bulman, STACK COMPUTERS; D.M. Bulman, STACK COMPUTERS, AN INTRODUCTION; R.P. Blake, EXPLORING A STACK ARCHITECTURE, J. Couch, T. Hamm, SEMANTIC STRUCTURES FOR EFFICIENT CODE GENERATION ON A STACK MACHINE; F.G. Duncan, STACK MACHINE DEVELOPMENT: AUSTRALIA, GREAT BRITAIN, AND EUROPE.

Varie

E.B. Daly, MANAGEMENT OF SOFTWARE DEVELOPMENT, IEEE Transactions on Software Engineering, vol. SE-3, n.3, maggio 1977, pagg. 230-242.

This paper describes four major aspects of software management: development statistics, development process, development objectives, and software maintenance. The control of both large and small software projects is included in the analysis".

S.H. Zweben, A STUDY OF THE PHYSICAL STRUCTURE OF ALGORITHMS, ibidem, pagg. 250-258.

"A theory of the structural composition of an algorithm is presented wich allows the frequencies of occurrence of the individual operators and operands to be estimated. It provides justification for some recent hypotheses which suggest certain functional relatioships between properties of algorithms".

E.W. Dijkstra, PROGRAMMING: FROM CRAFT TO SCIENTIFIC DISCIPLINE, International Computing Symposium 1977, Liege, 4-7 aprile 1977, pagg. 23-30

A.V. Gelder, STRUCTURED PROGRAMMING IN COBOL: AN APPROACH FOR APPLICATION PROGRAMMERS, Comm. ACM, Vol.20, n.1, 2-12 (gennaio 1977)
"...top-down program design is implemented through the use of structured flow-charts, disciplined specifications, and step by step verification..."

Ingegneria del software - Libri

T. GILB, G.M. Weinberg, HUMANIZED INPUT - TECHNIQUES FOR RELIABLE KEYED INPUT, Winthrop Publishers, Inc., Cambridge, Massachusetts, pgg. 283 (1977)

T. Gilb, DATA ENGINEERING, Student-litteratur, Lund (Sweden) pgg. 204 (1976)
"... a collection of new techniques, and of old techniques which are presented in a new light..."

J.K. Hughes, J.I. Michtom, A STRUCTURED APPROACH TO PROGRAMMING, Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, pgg. 264 (1977)
"The purpose of this book is to convey sound ideas and effective techniques needed for a structured approach. .... This book is designed for anyone who is familiar with the basic concepts of programming..."

Ingegneria del software - Tesi di PhD

P.F. Gehring Jr, A QUANTITATIVE ANALYSIS OF ESTIMATING ACCURACY IN SOFTWARE DEVELOPMENT (Agosto 1976, Texas A&M University)
"This research quantitatively examines the estimating accuracy of over 5000 standardized resource consuming activities from 39 software development projects of various size which were accomplished at the U.S. Air Force Data Systems Design Center...."

R.F. Bridge, SOFTWARE REDUNDANCY THROUGH FORMAL PROCEDURE INTERFACE SPECIFICATIONS, (University of Texas at Austin, Agosto 1976).

A.J. Turner, ITERATIVE ENHANCEMENT: A PRACTICAL TECHNIQUE FOR SOFTWARE DEVELOPMENT (Univeristy of Maryland, 1976)

P.A. Carr, NAME MANAGEMENT IN THE CONSTRUCTION OF LARGE PROGRAMS, (Iowa State University, 1976)
"... an investigation into the question of extending block structure to provide additional, perhaps more natural, ways of decomposing a large program, and to provide mechanisms for succintly describing the communication between the parts of the decomposed program ..."

G.C. Roman, PROGRAM CONTROL RESTRUCTURING: A SOFTWARE ENGINEERING METHODOLOGY, (University of Pennsylvania, 1976)
"... the program structure proposed requires the use of a control block and a set of modules, among which the initialization module is distinguished..."

Honeywell

Honeywell Information Systems Italia

Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Via Viotti, 5 - Milano